# Das Anwendungsprotokoll MQTT im Internet of Things

Bachelor Thesis

im Studiengang

Medien und Informationswesen


von Mario Sallat

betreut durch

Prof. Dr. Tom Rüdebusch

Zweitbetreuer:

Prof. Dr. Volker Sänger


im Wintersemester 2017/2018


Hochschule Offenburg

Fakultät M+I


Hochschule Offenburg
offenburg.university

M+I Medien und Informationswesen

# Inhaltsverzeichnis

# 1. Vorwort

Das *Internet of Things* ist inzwischen ein allgegenwärtiger und geläufiger Begriff in der heutigen Gesellschaft. Anbieter von Smart-Home-Anwendungen überbieten sich immer wieder mit neuen Innovationen. Auch in der Industrie hat das Internet of Things längst Einzug erhalten (*Industrial Internet of Things*). Das autonome vernetzte Auto begeistert nun nicht mehr einzig allein die Fans von Filmen wie *Zurück in die Zukunft II*, sondern ist im Alltag angekommen. Die Popularität von Smartphones und die Abhängigkeit zu diesen eröffnet die Möglichkeit, eine Schnittstelle von Maschine zu Mensch herzustellen, die fast immer verfügbar ist [1]. Der Begriff des *Ubiquitous Computing*, also die ständige Umgebung von Computern, Sensoren und Aktoren ist nicht neu. Diesen Begriff prägte schon Mark Weiser im Jahr 1991, als er davon sprach, wie intelligente Technik in derart prägnanter Weise unseren Alltag beeinflussen würde [2].

> *"The real power of the concept (e.g. ubiquitous computing) comes not from any one of these devices; it emerges from the interaction of all of them."* [2]

Die enorme Anzahl der mittlerweile mit dem Internet vernetzten Geräte gibt ihm recht. In Zahlen ausgedrückt: Momentan sind über 20 Milliarden IoT-Devices mit dem Internet verbunden. Schon im Jahr 2022, so wird geschätzt, soll sich die Anzahl auf über 40 Milliarden verdoppeln [3]. Über die letzten Jahre hat sich gezeigt, dass das Wachstum exponentiell ist, noch im Jahr 2003 lag die Zahl bei nur etwa 500 Millionen Einheiten [4]. Diesen Geräten wird im Jahr 2022 einen Anteil von 45% am gesamten Internet Traffic zugeschrieben [5] [6].

Diese ganzen Zahlen deuten alle in eine Richtung: In Zukunft wird der IoT-Sektor und die damit einhergehende *Machine-to-Machine-Kommunikation* (M2M) weiterhin ein signifikantes Wachstum aufweisen. Netzwerke müssen deswegen erweitert und die Datenübertragungsraten den Datenmengen angepasst werden, weil der *Internet Traffic* weiter kontinuierlich steigt. Neben dem Ausbau von Mobilfunk der fünften Generation *(5G)* und der Ausweitung von *Adress Space* durch *IPv6* muss die Infrastruktur des Internets auf die Herausforderungen und Bedürfnisse des Internet of Things ausgelegt werden. Eine Antwort auf die steigenden Anforderungen kann sein, neue leichtgewichtige Techniken, Standards und Schnittstellen zu entwickeln, die kostengünstig und ressourcensparender sind [7]. Ein Beispiel wäre leichtgewichtigere Übertragungsprotokolle anstatt des viel verwendeten HTTP einzusetzen, was für eine Verringerung der Datenmengen sorgen könnte.

Ein in letzter Zeit immer populärer werdendes Protokoll ist das *Message Queue Telemetry Transport (MQTT)* Protokoll. Es verspricht ein extrem leichtgewichtiges Protokoll für M2M-Kommunikation beim Nachrichtentransport auf Anwendungsebene zu sein, das vielseitig im IoT eingesetzt werden soll. Aufgrund seiner effizienten Nachrichtenvermittlung soll es ebenfalls für mobile Anwendungen geeignet sein [8].

Diese Thesis soll einen umfassenden Überblick über das MQTT-Protokoll im Einsatzgebiet Internet of Things geben. Das Ziel ist es, in einem theoretischen Teil den State of the Art des Protokolls darzustellen und dessen Vorteile hervorzuheben. Hierfür wird in

Kapitel 2 definiert, was der Begriff Internet of Things bedeutet. Im 3. Abschnitt wird MQTT vorgestellt. Im praktischen Teil dieser Thesis wird in Kapitel 4 ein MQTT-Testbett konzipiert und anschließend prototypisch implementiert. Darauf aufbauend wird eine Dokumentation erstellt, die als Vorlage für einen Laborversuch für die Lehrveranstaltung *Interaktive verteilte Systeme* an der *Hochschule Offenburg* dienen soll. Im weiteren Verlauf wird zusammenfassend das implementierte System unter Berücksichtigung der gewonnenen Erkenntnisse und der zuvor gestellten Anforderungen an das System evaluiert.

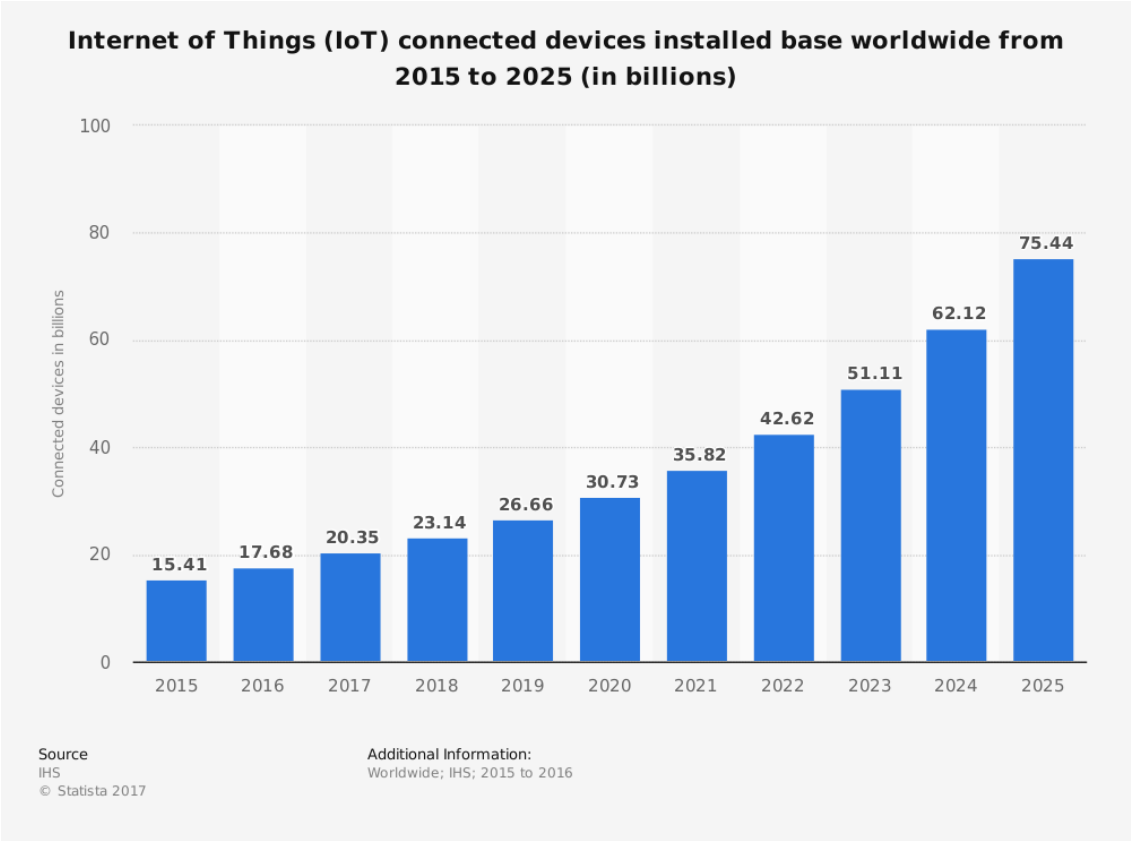



*Abbildung 1: mit dem Internet verbundene IoT-Devices [3]*

# 2. Internet of Things

## 2.1. Definition


Das *Internet of Things (Internet der Dinge, IoT)* beschreibt Systeme aus physikalischen Objekten, sogenannte T*hings*, die in Software, Netzwerke und Elektronik eingebunden sind [9]. Laut der deutschen EU-Ratspräsidentschaft im Jahr 2007 wird die Definition wie folgt festgelegt: „[Internet der Dinge bedeutet] die technische Vision, Objekte jeder Art in ein universales digitales Netz zu integrieren." [7]. Diese intelligenten Objekte sammeln und empfangen Daten und tauschen sie untereinander aus. Dabei kann die Kommunikation, wie der Name IoT schon sagt, auch über das globale Internet stattfinden.

Drei Eigenschaften zeichnen die Objekte im IoT meist aus: Neben der *Allgegenwärtigkeit (Ubiquitous Computing)* sind diese *unsichtbar* in unserer Umgebung integriert und weisen durch automatisierte Prozesse in der Regel eine hohe *Autonomie* auf [7]. Im Kern ist der wichtigste Bestandteil der ständige Austausch von Nachrichten, auch *Machine-to-Machine-Kommunikation* genannt.

Durch die fortschreitende Digitalisierung entstehen weiterhin immer mehr *intelligente Umfelder*, in denen es Verbindungen zwischen der physischen Welt und der digitalen Welt des Internets gibt [7]. Dabei ist das Internet of Things ein Überbegriff für verschiedene Bereiche, die sich in folgende Sektoren untergliedern lassen: Im Endanwender-Markt, also bei dem Kontakt mit Objekten im alltäglichen Leben, lassen sich beispielsweise intelligente Haustechnik *(Smart Home, Digital Life)*, autonome Fahrzeuge (*Future Mobility*), intelligente Verkehrssysteme (*Smart City*) oder schlaue Energieversorgung (*Smart Meters, Smart Grid*) nennen [7]. Auf der anderen Seite steht der Begriff des *Industrial Internet of Things (IIoT, Industrie 4.0)*. Hier liegt das Hauptaugenmerk auf der smarten Vernetzung von Maschinen und Robotern, der Verbesserung von Produktionsabläufen in Fabriken durch vernetzte Sensorik und einer weiteren Digitalisierung und Bereitstellung von Produkten, Dienstleistungen und Daten *(Smart Factory, Platform as a Service, Big Data, Cloud Infrastruktur)* [10].

## 2.2. Geschichte

Den Begriff „Internet of Things" verwendeten Forscher des *Massachusetts Institute of Technology (MIT)* erstmals Ende des letztens Jahrtausends. In erster Linie arbeiteten Kevin Ashton und David L. Brock dort an Forschungsthemen wie Techniken zur Identifizierung, Datenerfassung, Datenerhebung und Datenübertragung *(Auto-ID)* [11]. Diese beinhalten beispielsweise Barcodes, Sensoren, Smart Cards, Spracherkennung und seit dem Jahr 2003 auch *Radio Frequency Identification (RFID)* [12]. Diese am MIT entwickelten Technologien gelten als die wichtigsten Voraussetzungen und als Grundstein für die Machine-to-Machine-Kommunikation. Gerade RFID wird eine wichtige Rolle zugesprochen.

An einer grundlegenden Technologie arbeiteten die Forscher, bei der Computer automatisch einen kompletten Herstellungsprozess eines Produktes vom Fließband bis zur Auslage auf der Verkaufsfläche automatisiert überwachen konnten. Daraus ergab sich die Zielsetzung, eine Vernetzung von Computern und Objekten im globalen Internet vorzunehmen, die nicht nur aus Hardware und Software, sondern auch aus Protokollen und Beschreibungssprachen für die Objekte bestand. Noch bestand das Internet of Things jedoch nicht in der Form, wie es heutzutage bekannt ist. Das hatte den Grund, dass die Zahl der Devices im Internet in Bezug auf die Weltbevölkerung noch relativ gering war (im Jahr 2003 ∅ 0,08 Geräte/Person), als dass von Ubiquitous Computing gesprochen werden könnte [11].

In den nächsten Jahren wurde das Internet of Things durch verschiedene Veröffentlichungen [13] [14] [15] [16] einer breiteren Menge bekannt. Mit der Einführung des iPhones im Jahr 2007 wurde auch die Zahl der mit dem Internet verbundenen Geräte weiter erhöht. Der Anstieg der Verbreitung gerade im Consumer-Bereich sorgte dafür, dass die Anzahl der Geräte um das Jahr 2008 die Anzahl der Weltbevölkerung überholte (im Jahr 2010 ∅ 1,84 Geräte/Person). An dieser Stelle wird gerne von der *Geburt des Internet of Things* gesprochen [11].

Seit einiger Zeit wird das Potential durch das Internet of Things auch von Politik und Wirtschaft erkannt. Im Jahr 2009 wurde von der Europäischen Union ein Aktionsplan für die Entwicklung von IoT veröffentlicht, der eine Vielzahl von Forschungsprojekten zu IoT vorsah [17]. In Deutschland wird die Forschung zu IoT mittlerweile auch durch die Ministerien für Bildung und Forschung sowie für Wirtschaft und Technologie gefördert [7].

## 2.3. Vor- und Nachteile

Wie bereits oben aufgeführt, ist IoT eine Technologie, die schon in sehr vielen Lebensbereichen Einzug genommen hat. Die prognostizierten Zahlen zu den Geräten sagen zusätzlich ein enormes Wachstum in diesem Bereich voraus. Aus diesem Grund hat IoT ein sehr hohes ökonomisches Potential für Unternehmen. Für fast jede Art von Problemen der Industrie können IoT-Lösungen bereitgestellt werden, was einen hohen Faktor an Prozessoptimierung mit sich bringen wird. Kosten können gesenkt und die Effizienz bei Vertriebs-, Produktions- und Service-Prozessen gesteigert werden. Gerade dem Bereich Big Data kommen die hohen Datenmengen, die bei der hohen Anzahl von Objekten im IoT anfallen, zugute. Im privaten Bereich wird die Lebensqualität durch automatische Prozessabläufe verbessert, indem Menschen Entscheidungen von Systemen abgenommen werden und diese dennoch die Kontrolle beispielsweise per App auf dem Smartphone über ihr intelligentes Zuhause behalten. Gerade auch im oft sehr hoch belasteten Gesundheitswesen werden Aufgaben mit der Zeit von IoT abgedeckt sein, was eine Entlastung von Ärzten bedeuten kann, weil zum Beispiel der Blutdruck von älteren Menschen automatisch von zuhause an den Arzt übermittelt würde, ohne dass diese die Patienten bei sich in der Praxis empfangen müssen.

Der größte Vorteil, dass so zahlreiche Geräte miteinander über das Internet vernetzt werden können, kann jedoch auf der anderen Seite auch zu einem großen Risiko werden. Denn mit dem Internet verbundene Geräte, auf denen ein Betriebssystem läuft,

stellen in den meisten Fällen auch eine potentielle Gefahr dar, kompromittiert zu werden. Schwachstellen können in einem IoT-System unterschiedlicher Art sein. Gerade bei lebenswichtigen Systemen in Bereichen wie Smart Health oder intelligente Stromversorgung ist es essentiell, dass diese Systeme ständig am Laufen sind, um beispielsweise einen reibungslosen Ablauf in einem Kraftwerk oder Krankenhaus zu gewährleisten. Wenn nun bei einem Cyber-Angriff durch eine *DDoS Attacke (Distributed Denial-of-Service)* Teile von Systemen lahmgelegt würden, hätte das fatale Konsequenzen zur Folge. In der Vergangenheit wurden bereits Angriffe festgestellt, bei denen in erster Linie ältere IoT-Geräte aufgrund mangelnder Sicherheitsvorkehrungen kompromittiert worden sind, zu einem *Botnetz* zusammengeschlossen wurden und selbst DDoS-Attacken durchführten. So passierte es bei der bekannten *2016 Dyn Cyberattack* am 21. Oktober 2016, als durch den Angriff auf das *Dynamic Name System (DNS)* weite Teile des Internets in den USA lahmgelegt wurden [18]. Nicht zu vernachlässigen sind auch Datenschutz,- und Sicherheits-Aspekte der Nutzer. Im Jahr 2013 wurde bekannt, dass ein mit dem Internet verbundenes Babyphone einem Angreifer ermöglichte, Konversationen in einer Wohnung mitzuhören und somit ein zwei Jahre altes Kind zu belauschen [19]. An dieser Stelle besteht Handlungsbedarf, gerade wenn es um Themen wie Vertraulichkeit, Authentizität und Integrität geht, müssen eindeutige Verantwortlichkeiten und Haftungsregeln festgelegt werden [7].

## 2.4. Architektur

### 2.4.1. Protokolle

Vor dem Aufbau einer Verbindung von Geräten über ein Netzwerk muss sichergestellt werden, dass beide Geräte durch einen zuvor festgelegten Standard diese Verbindung zueinander unterstützen. Ermöglicht wird dieser Schritt durch standardisierte *Netzwerkprotokolle*. Diese Protokolle entsprechen der Sprache, über die die Geräte miteinander kommunizieren. Hier werden die Regeln, also Handlungen bei dem Senden und Empfangen von Nachrichten, Formate und der genaue Ablauf der Netzwerkkommunikation, unter anderem die Reihenfolge des Nachrichtenaustauschs, festgelegt [20].

### 2.4.2. Protokollschichten im Internet

Die Protokolle werden zur besseren Strukturierung nun unterschiedlichen *Netzwerkschichten (Layers)* zugeordnet. Wenn ein Gerät mit der entsprechend gleichen Schicht eines anderen Gerätes kommuniziert, wird von *horizontaler Kommunikation* gesprochen. Dabei stellt eine Schicht im sogenannten *Protokollstapel* (siehe Abbildung 2, Seite 6) spezielle Funktionalitäten bereit, die als *Dienst* oder *Service* beschrieben werden. Bei der *vertikalen Kommunikation* zwischen den Schichten wird beim Durchreichen die Information der tieferliegenden Schicht verpackt oder entpackt. Hierbei wird von *Encapsulation* gesprochen. Vertikale Kommunikation findet beispielsweise dann statt, wenn beim Übertragen einer Nachricht eines Anwendungsprogramms von einem Computer zu einer Anwendung eines anderen Computers über die Protokollschichten die Information nach unten zur physikalischen Übertragungsschicht durchgereicht wird, dann durch das Übertragungsmedium übertragen wird und anschließend auf dem empfangenden Computer die Schichten in unterschiedlicher Reihenfolge wieder bis zur An-

wendungsschicht durchlaufen werden [21]. Besonders drei Modelle sind bei der Netzwerkkommunikation über das Internet bedeutsam, bei denen sich die Schichtenarchitekturen einander ähneln. Die Schichtenmodelle *TCP/IP-Referenzmodell, Hybrides Referenzmodell* und das *OSI-Referenzmodell* werden in Abbildung 2 dargestellt. Der *TCP/IP-Protokollstack* ist beispielhaft für die Anwendungsprotokolle MQTT, HTTP und CoAP aufgeführt.

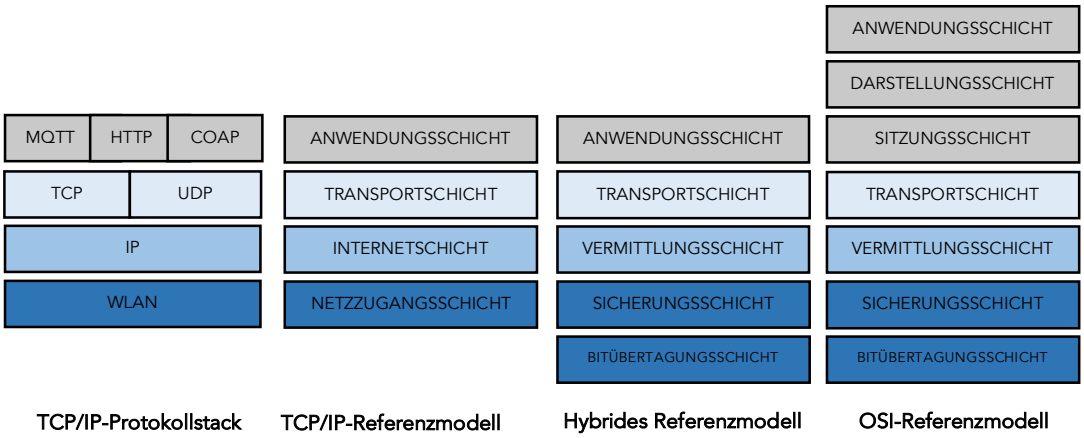



*Abbildung 2: Vergleich der Referenzmodelle [22]*

### Bitübertragungsschicht

In dieser Schicht, auch *Physical Layer* genannt, werden physikalische und technische Merkmale des Übertragungsmediums definiert. An dieser Stelle werden die Informationen über einen physikalischen Kanal per Nullen und Einsen übertragen. Zu den Aufgaben der *Bitübertragungsschicht* gehört festzulegen, in welche Richtung die Übertragung stattfindet, wie der Aufbau und das Beenden einer Verbindung geschieht und welche Art von Modulation verwendet wird, also in welche Signale der gesendete Bitstrom konvertiert wird. Bekannte Protokollstandards sind hier beispielsweise *DSL (Digital Subscriber Line)* oder *IEEE 802.11 PHY* [21] [22].

### Sicherungsschicht

Der *Data Link Layer* hat die Aufgabe, bei dem Auftreten von Bitfehlern bei Punkt-zu-Punkt Verbindungen mehrerer Netzwerkkomponenten innerhalb der Bitübertragungsschicht, diese zu korrigieren und so eine zuverlässige Verbindung zu gewährleisten. Die Protokolle der *Sicherungsschicht* regeln zusätzlich den Zugriff auf das physikalische Übertragungsmedium. Im Einzelnen bedeutet das, dass bei dem sendenden Gerät die von der darüber liegenden Schicht *(Vermittlungsschicht)* kommenden Pakete in sogenannte *Rahmen (Frames)* verpackt, mit der definierten Zuverlässigkeit übertragen, und eventuelle Fehler durch Fehlererkennungsverfahren wie Prüfsummenberechnung behoben werden. Die Sicherungsschicht definiert außerdem die physischen Adressen, um eine Zustellung der Frames zu gewährleisten. Wichtige Protokolle dieser Schicht sind *IEEE 802.3 (Ethernet)*, *IEEE 802.11 MAC (Medium Access Control)* und *PPPoE (Point-to-Point Protocol over Internet)* [21] [22].

### Vermittlungsschicht

In der *Vermittlungsschicht* (engl.: *Network Layer*) werden Mittel zur Verfügung gestellt, mit Hilfe derer Datenpakete vom Sender zum Empfänger über mehrere logische Netzwerke hinweg gesendet werden können. Für diese Funktionalität werden logische Adressen *(IP-Adressen)* benötigt, die im Network Layer definiert werden. Der Transfer von Datenpaketen über physische Übertragungsabschnitte hinweg wird auch *Internetworking* genannt. Die logischen Netzwerke werden durch *Router* begrenzt. Diese sind auch dafür zuständig, dass auf dieser Schicht die Datensequenzen, die eine variable Länge aufweisen können, durch das sogenannte *Routing* vom Sender zum Empfänger gelangen. Das meist verwendete Protokoll dieser Schicht ist das verbindungslose *Internet Protocol (IP)* [21] [22].

### Transportschicht

Die *Transportschicht*, auch *Transport Layer* genannt, stellt den Endsystemen verschiedener Netzwerke eine Ende-zu-Ende-Verbindung zur Verfügung. Hier werden *Segmente* von Prozessen verschiedener Geräte transportiert. Dafür werden die anfallenden Daten der darüber liegenden Schicht beim Sender an dieser Stelle in diese sogenannten Segmente verpackt. Im Gegensatz zu den Diensten der Vermittlungsschicht ist der Transportdienst eines Protokolls der Transportschicht ein zuverlässiger Dienst. Hier werden die Eigenschaften definiert, dass eine zuverlässige und sichere Datenverbindung stattfinden kann. Eine Adressierung der Prozesse beim End-Anwender findet in diesem Layer über Portnummern statt. Unterschiedliche Protokolle bieten hierbei verschiedene Kommunikationsformen zwischen Geräten an. Wenn erwünscht ist, dass ein Segment beim Empfänger verlässlich ankommt oder die korrekte Reihenfolge der Segmente wichtig ist, muss ein Protokoll mit einer *verbindungsorientierten Kommunikation* verwendet werden. Vor dem Datenaustausch wird hier eine Verbindung aufgebaut und danach wieder abgebaut. Ein Vertreter, der einen verbindungsorientierten Dienst liefert, ist das *Transport Control Protocol (TCP).* Im Gegensatz dazu baut das *verbindungslose Kommunikationsprotokoll User Data Protocol (UDP)* keine Verbindung vor der Datenübertragung auf. Anwendung findet dieses Protokoll, wenn kein Fokus auf der Kontrolle der Segmente liegt oder ein höherer Datendurchsatz gewünscht ist [21] [22].

### Anwendungsschicht

Die *Anwendungsschicht* stellt Services für Anwendungsprogramme bereit, die eine Netzwerkverbindung nutzen wollen. Diese Dienste ermöglichen es, dass die Nutzung des Kommunikationssystems über den Stack vereinfacht wird, und der Endanwender beziehungsweise der Programmierer nur noch die Protokolle der Anwendungsschicht als Schnittstelle nutzen muss. Die Aufgaben, die den Anwendungsprotokollen zugesprochen werden, sind beispielsweise die Synchronisation der Kommunikation und die Identifizierung der Kommunikationspartner, beziehungsweise deren Anwendungen. Das Anwendungsprogramm an sich gehört nicht zur Anwendungsschicht. Applikationen, die die Anwendungsschicht nutzen, sind zum Beispiel Browser, E-Mail-Clients oder Datentransferprogramme. Zu den verwendeten Anwendungsprotokollen gehören das *Hypertext Transfer Protocol (HTTP), Secure File Transfer Protocol (SFTP), Secure Shell (SSH), Simple Mail Transfer Protocol (SMTP), Constrained Application Protocol (CoAP)* oder *Message Queue Telementry Transport (MQTT)* [21] [22].

### 2.4.3. 5-Schichten-Modell der IoT-Architektur

Ein mögliches Modell für eine Architektur im Internet of Things sollte, wie in den vorherigen Kapiteln festgestellt wurde, es ermöglichen, die sehr hohe Anzahl von mehreren Milliarden Objekten flexibel miteinander über das Internet verbinden zu können. IoT bringt nicht nur die Anforderung mit sich, dass damit ein hohes Datenaufkommen produziert wird und darum verarbeitet werden muss, sondern es müssen ebenso Sicherheitsaspekte, sowie eine gute Skalierbarkeit, hohe Zuverlässigkeit und Eigenschaften für *Quality of Service (QoS)* beachtet werden [5] [23].

Mittlerweile gibt es für IoT eine Mehrzahl unterschiedlicher Projekte, die versuchen, eine allgemein gültige IoT-Architektur zu präsentieren. Ein prominenter Vertreter ist hier *IoT-A* [24], das auf dem *IoT Reference Model* [4] und dem *Architectural Reference Model (ARM)* basiert und mithilfe unterschiedlicher Business Szenarios und den dazugehörigen Stakeholdern entwickelt wurde [25].

Das Basis-Modell, auf dem die meisten IoT-Schichtmodelle aufbauen, ist die *3-Schicht-Architektur*. Diese besteht aus dem *Application-*, dem *Network-*, und dem *Perception-Layer*. Auch das neuere *5-Schichten-Modell* beruht darauf und erweitert dieses um den *Middleware-* und *Business-Layer* (siehe Abbildung 3). Der Aufbau ist ähnlich des Konzeptes des Hybriden Referenzmodells beim Internet-Protokollstapel und baut im Wesentlichen nur auf einzelnen Komponenten desselben auf. Der TCP/IP-Protokollstack ist in diesem Fall bei dem 5-Schichten-Modell dem Network-Layer zuzuordnen. Die 5-Schichten-Architektur ist laut Al-Fuqaha ein sehr geeignetes Modell für IoT-Anwendungen [5]. Im Folgenden werden die einzelnen Schichten dieses Modells und dessen Aufgaben beschrieben.

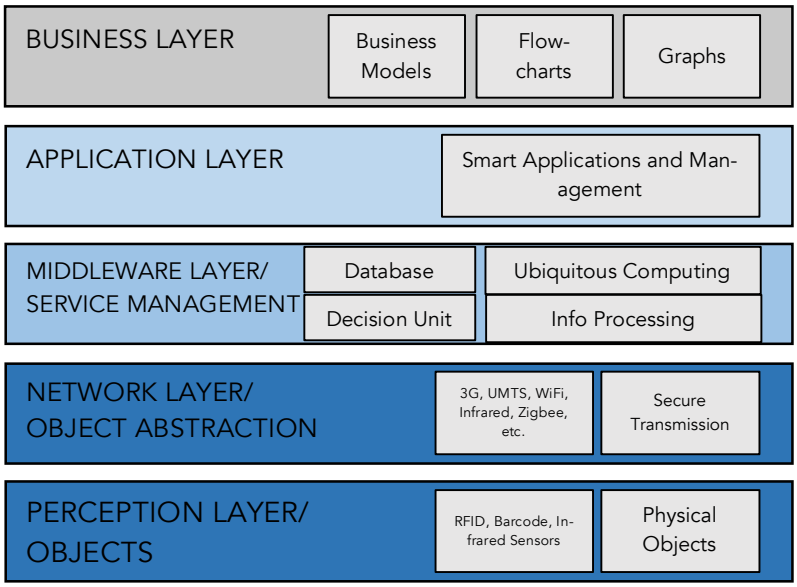



*Abbildung 3: IoT Architektur [23]*

#### Perception Layer

Die unterste Schicht beinhaltet die physischen Objekte in einem IoT-Netzwerk. Deswegen ist diese Schicht auch als *Objects Layer* oder *Device Layer* bekannt. An dieser Stelle haben *Sensoren* die Aufgaben, Informationen zu erfassen und *Aktoren* führen Aufgaben

durch. Die Aufgaben der Objekte können sein: Messung von Temperatur und Luftfeuchtigkeit, Detektieren der Orientierung und Ortung oder Messen von Geschwindigkeit und Beschleunigung. Eine weitere Aufgabe dieser Schicht ist die Identifizierung und Konfiguration von heterogenen Objekten, zum Beispiel über RFID. Nach der Informationserfassung und der Digitalisierung derselben wird sie an den Network Layer weitergereicht [23] [5] [26].

### Network Layer

Die Daten, die von den Objekten in vorherigen Schicht anfallen, werden mithilfe des *Network Layer*, auch *Transmission Layer* oder *Object Abstraction Layer* genannt, auf einem sicheren Kommunikationsweg, der kabellos oder kabelgebunden sein kann, zur nächsten Schicht transportiert. Gängige Übertragungsmedien können hier mobile Netze wie *UMTS*, *3G*, *4G* oder Netzwerke und Techniken wie *WLAN*, *Infrarot*, *ZigBee* oder *LPWAN* sein. Die Hauptfunktionalität besteht also im Transport der Informationen. Ein gängiges Protokoll dieser Schicht ist beispielsweise *IPv6 (Internet Protocol version 6)*, das für die Adressierung von Milliarden Geräten verantwortlich ist [23] [5] [26].

### Middleware Layer

Die Schichten der verschiedenen Geräte benutzen im IoT unterschiedliche Protokolle und bieten somit auch differenzierte Dienste an. Im *Middleware Layer*, oder auch *Service Management Layer* wird sichergestellt, dass die Services der Netzwerkprotokolle reibungslos miteinander kommunizieren. Voraussetzung ist, dass die Dienste der Geräte, die miteinander kommunizieren, vom gleichen Typ sind. Weiterhin findet in dieser Schicht die Verbindung zu möglichen Datenbanken statt. Hauptfunktionalitäten an dieser Stelle sind zusätzlich *Cloud Computing*, *Ubiquitous Computing* und intelligente Prozessabläufe. Für heterogene Objekte werden hier Informationen intelligent verarbeitet, was die Grundlage für automatische Entscheidungen setzt [23] [5] [26].

### Application Layer

Die Aufgabe des *Application Layer* beruht auf den Daten, die vom Perception Layer aufgenommen worden sind und im Middleware Layer verarbeitet wurden. Hier werden die Dienste mithilfe einer Applikation angeboten, die vom Benutzer angefordert werden. Spezifische Anwendungen im Bereich IoT können Smart Health, Smart Home oder intelligente Transportsysteme sein. Wenn der Nutzer beispielsweise eine Temperaturabfrage startet, werden die Informationen in dieser Schicht bereitgestellt. Dafür wird meist ein Nutzer-Interface implementiert, an dem der End-Nutzer mit dem System interagieren kann und die Informationen einsehen kann, die für ihn relevant sind [23] [5] [26].

### Business Layer

Der *Business Layer* ist für das Management des gesamten IoT-Systems verantwortlich. Die Daten jeder System-Aktivität, der Anwendungen, der verwendeten Services und der restlichen vier Schichten werden hier im Rahmen des verwendeten Business-Modells anhand von Graphen, Flowcharts und Tabellen bereitgestellt. Der Sinn dahinter ist, neue Elemente eines Systems mithilfe des Business-Modells zu entwickeln, indem die Daten in dieser Schicht gesammelt, analysiert, dargestellt und evaluiert werden. Durch die Analyse von *Big Data* wird hier die Grundlage für intelligente automatische Entscheidungen gelegt. Weiterhin werden an dieser Stelle die tatsächlichen Outputs einer

Schicht mit den erwarteten Outputs verglichen, um Verbesserungen und optimierte Erweiterungen im System zu erzielen [23] [5] [26].

## 2.5. IoT-Standards des Network Layer

Machine-to-Machine-Kommunikation ist das Schlüsselwort, wenn es um den Austausch von Daten im Internet of Things geht. In den verschiedenen Bereichen wie Smart Cities, Smart Homes, Smart Factories oder *Sensornetzwerken (WSNs)* stehen bei der Wahl des Kommunikationskanals für die smarten Geräte unterschiedliche Infrastruktur-Technologien zur Verfügung. In einem WSN werden mit Sensoren ausgestattete Devices *(Sensorknoten)* kabellos miteinander verbunden. Gängige Methoden für den Datenaustausch sind die klassischen Verbindungen über Kabel *(DSL, Ethernet)* und kabellose Technologien wie *WLAN* und *Bluetooth* oder *Low Power Wide Area Networks* wie *LoRaWAN*, Narrowband *IoT* oder der klassische Mobilfunk *(LTE, 4G, 5G)*. Diese Infrastruktur-Techniken sind in dem 5-Schichten-Modell der IoT-Architektur dem *Network Layer* zuzuordnen (siehe Abbildung 2, Seite 6). Nachfolgend soll ein Überblick über die herkömmlichen kabellosen Technologien für IoT gegeben werden.

### WLAN (IEEE 802.11)

*Wireless Local Area Network (WLAN)* ist ein Kommunikationsstandard der *IEEE 802.11* Normenfamilie. Es beschreibt ein lokales Netzwerk in den Frequenzbereichen um 2,4 GHz und 5 GHz. Bei diesem Netzwerkprotokoll werden hohe Sendeleistungen erreicht, sogar Datenübertragungsraten im Gigabit-Bereich (3,5 Gbit/s) werden je nach Frequenzband (5 GHz) und Modulationsverfahren (QAM256) im 802.11ac Standard erreicht [27]. Den hohen Sendeleistungen und Reichweiten steht bei dieser Technologie jedoch ein relativ hoher Stromverbrauch gegenüber. Geräte in einem WLAN-Netzwerk können über zwei verschiedene Modi miteinander verbunden werden. Im weiter verbreiteten *Infrastruktur-Modus* wird eine Basisstation benötigt, bei der die Endgeräte sich mit ihrer *MAC-Adresse* anmelden müssen. Damit der Aufbau der Verbindung ermöglicht wird, sendet der *Access-Point* in regelmäßigen Abständen sogenannte *Beacon Frames*, die unter Anderem den *Netzwerknamen (SSID)*, die Verschlüsselung und die unterstützten Übertragungsraten beinhalten. Bei dem *Ad-hoc-Modus* hingegen ist eine vermaschte Netztopologie vorhanden. Jedes Endgerät kann mehrere Verbindungen eingehen, die Geräte sind direkt miteinander verbunden [22].

### Bluetooth (IEEE 802.15.1)


*Bluetooth* wurde schon 1994 von der schwedischen Firma *Ericsson* entwickelt und ist damals wie heute für drahtlose Kommunikation zwischen Geräten auf kurze Distanzen ein weit verwendeter Standard, der ursprünglich dafür gedacht war, eine Alternative für kabelgebundene Kommunikation darzustellen [28]. Mittlerweile wird die Entwicklung der neuesten Bluetooth-Protokolle von der *Bluetooth Special Interest Group (SIG)* vorangetrieben. Die aktuelle Version ist *Bluetooth 5.0*. Die Versionen vor *Bluetooth 2.1* werden als *Classic Bluetooth* beschrieben. Mit *Bluetooth 4.0* wurde ebenfalls *Bluetooth Low Energy (Bluetooth LE)* eingeführt, um auf dem Markt der leichtgewichtigen energiesparenden Protokolle Boden zu fassen. Der Betrieb von Bluetooth findet im Frequenzblock von 2,402-2,480 GHz statt. Die Topologie besteht meist aus *Piconetzen*, die aus maximal 255 Geräten bestehen können. Acht Devices können hier nur zu einem Zeitpunkt aktiv sein, die restlichen Geräte sind passiv.

Das bedeutet, dass ein *Master*, der den Datenverkehr und die aktiven Verbindungen koordiniert, mit sieben weiteren Geräten *(Slaves)* kommunizieren kann. Bluetooth-Geräte sind so konfiguriert, dass sie in mehreren Piconetzen aktiv sein, jedoch nur in einem von diesen die Rolle als Master einnehmen können [22] [28].

Wie zuvor erwähnt kam mit Bluetooth 4.0 das Konzept des Bluetooth LE, um eine leichtgewichtigere Kommunikation zu ermöglichen. Die Zeit der Nachrichtenübermittlung wurde hier auf einen einstelligen Millisekunden-Bereich reduziert und es unterstützt Geschwindigkeiten bis zu einem Mbit/s. Devices, die Bluetooth LE unterstützen, werden mit dem Label *Bluetooth Smart* versehen. Diese Geräte sind nur mit Geräten kompatibel, die den *Bluetooth Smart Ready* Standard unterstützen (siehe Abbildung 4). Bluetooth Smart Ready ist wiederum abwärtskompatibel zu Bluetooth Classic Devices. Die Topologie unterscheidet sich bei Bluetooth LE dahingehend von Bluetooth Classic, dass ein physischer Kanal, also eine Master-Slave Verbindung, nun nur noch aus zwei Geräten besteht [28].

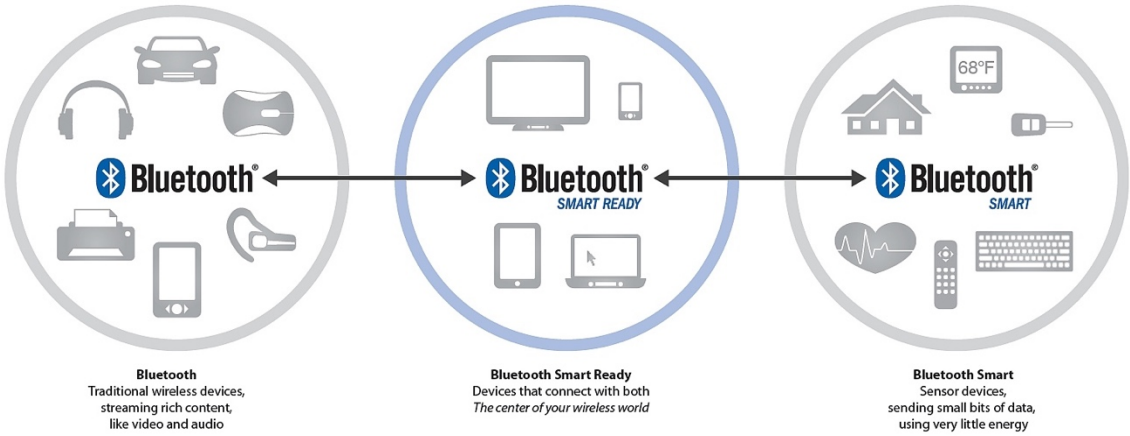



*Abbildung 4: Beziehung zwischen Bluetooth Smart und Bluetooth Smart Ready [59]*

Nach zwei weiteren Releases von *Bluetooth 4.1* und *Bluetooth 4.2* wurde 2016 Bluetooth 5.0 veröffentlicht. Von offizieller Stelle wird gesagt, dass die Entwicklungen an dieser Version vor allem im Fokus von Internet of Things stehen. Dieses Protokoll ist abwärtskompatibel bis Bluetooth 4.0 und somit auch Bluetooth LE und soll Geräten, die diesen Standard unterstützen, eine bis zur viermal größere Reichweite bescheren, sowie mit einer Übertragungsrate von zwei Mbit/s zweimal schneller als der letzte Standard sein. Eine weitere Verbesserung ist die Erkennung und das Verhindern von Interferenzen mit Technologien, die ein benachbartes Frequenzband benutzen. Zu den interferierenden Geräten gehören unter anderem das *2,4 GHz ISM-Band (Industrial, Scientific and Medical Band)* oder das *LTE-Band (Long Term Evolution Band)* [29].

Eine Neuerung, die mit Bluetooth 5.0 einhergeht, ist die *Bluetooth Mesh Specification*. Bluetooth Mesh ändert die bisherige Topologie von Bluetooth grundlegend, denn neben Punkt-zu-Punkt-Verbindungen können vermaschte Netzwerke aufgebaut werden [30]. Bei dieser Art von Netztopologie können alle Knoten miteinander verbunden werden, was die Reichweite von Bluetooth-Geräten vergrößert und das Risiko eines Systemausfalls verringert, da defekte Geräte umgangen werden können. Auch aufgrund des Supports des *Publish/Subscribe-Patterns* (siehe 3.3.1, Seite 21) eignet sich diese

Spezifikation, die im Juli 2017 in der Version 1 veröffentlich wurde, für IoT-Anwendungen [31].

### IEEE 802.15.4

Dieser Standard definiert die beiden untersten Schichten des OSI-Referenzmodells: Den Medium Access Control Layer (MAC) und den Physical Layer (PHY). *IEEE 802.15.4* beschreibt ein Kommunikationsprotokoll für *Low Rate Wireless Personal Area Networks (LR-WPANs).* Eigenschaften dieses Protokolls sind ein energiesparender Betrieb, niedrige Datenraten und hoher Datendurchsatz und es bietet eine zuverlässige Kommunikation, einen hohes Sicherheitslevel und kann auf vielen Plattformen betrieben werden. Bei den Betriebsfrequenzen von 2,4 GHz, 915 MHz und 868 MHz können jeweils Datenraten von 250 kbit/s, 40 kbit/s und 20 kbit/s erreicht werden [5] [32].

### 6LowPAN

Ein bekannter Standard einer Implementierung eines WSN ist *IPv6 over Low Power Wireless Personal Area Networks (6LowPAN).* Zuständig für dieses Kommunikationsprotokoll ist die Arbeitsgruppe der *IETF (Internet Engineering Task Force)*, die den Standard 2007 entwickelte. Bei diesem Verfahren wird der Standard-Header komprimiert, um IPv6-Pakete über IEEE 802.15.4-basierende Netzwerke zu schicken. Konkret heißt das, dass eine Adressierung von *Sensor Nodes* (Sensor-Devices) innerhalb eines WSN mithilfe des adressreichen IPv6-Protokolls vorgenommen werden kann. Die Datenpakete an sich werden bei diesem Protokoll fragmentiert, um das *IPv6 Maximum Transmission Unit (MTU, maximale Übertragungseinheit)* einzuhalten [5].

### ZigBee

*ZigBee* ist eine Spezifikation, die durch die 2002 gegründete *ZigBee Allianz* entwickelt wurde und auf dem IPv6-Protokoll basiert. Es baut den IEEE 802.15.4-Standard auf und erweitert diesen (siehe Abbildung 5, Seite 13). Mithilfe von Routing hat ZigBee die Eigenschaft, dass Datenpakete über mehrere Funkmodule hinweg geleitet werden können. Zusätzlich braucht ein über den ZigBee-Protokollstapel implementiertes System wenig Ressourcen auf den jeweiligen Plattformen, denn Speicherplatz und Rechenleistung sind auf der meist eingesetzten kostengünstigen Hardware begrenzt. Die ZigBee-Anwendung liegt eine Schicht über der Netzwerkschicht und kann über ein Framework implementiert werden. Vorteile sind lange Batterielaufzeiten, eine hohe Kompatibilität der Geräte untereinander und niedriger Overhead durch den sparsamen Einsatz von Steuerinformationen.

Geräte, die über dieses Protokoll miteinander kommunizieren, bilden ein LR-WPAN. Die Verwaltung der unterschiedlichen Geräte liegt bei dem sogenannten ZigBee-*Koordinator.* ZigBee basiert auf den zwei untersten Schichten des IEEE 802.15.4-Standards. Darüber liegt die Vermittlungsschicht, welche für das Routing verantwortlich ist. Hierfür werden *ZigBee-Router* benötigt. Die *ZigBee-Endgeräte*, welche den eingeschränktesten Funktionsumfang haben, müssen erst dem Router beitreten. Der Grund für den Vorteil eines niedrigen Energiebedarfs eines Endgerätes liegt unter anderem darin, dass dieses in einen Schlafmodus versetzt werden kann [32].

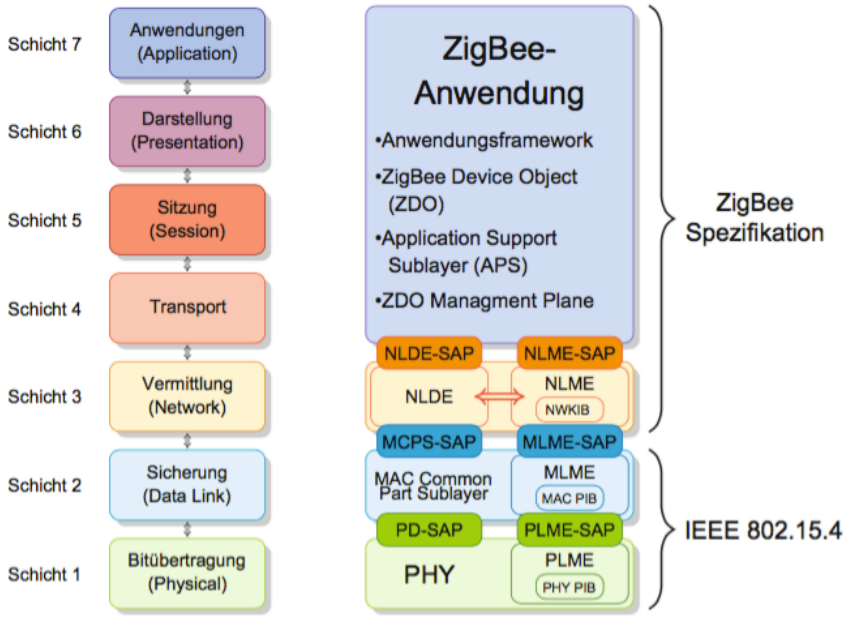



*Abbildung 5: OSI-Schichtenmodeel und IEEE 802.15.4/ZigBee im Vergleich [32]*

### Thread

Das für das Internet of Things und ganz speziell auf Hausautomatisierung zugeschnittene Netzwerkprotokoll *Thread* soll laut dem Hersteller einfach zu bedienen, sicher und energiesparend sein. Es baut wie ZigBee auf den 802.15.4-Standard auf, hat also schon den MAC und den PHY-Layer integriert. Diese sorgen unter anderem dafür, dass verbaute Nodes im System bis zu einem Jahr mit einer kleinen Batterie auskommen können. Die Typologie des kabellosen Netzwerks ist vermascht und ist bei Vorkommnissen, beispielsweise bei dem Ausfall eines Knotens, selbstheilend. Aufgrund der IPv6-Unterstützung sind die Geräte über IP adressierbar [33].

### Z-Wave

*Z-Wave* ist ein von der Firma *Sigma Designs* entwickeltes Kommunikationsprotokoll, das auf *RF-Kommunikation (Radio Frequency)* basiert. Der Hersteller wirbt damit, dass diese Technologie im Gegensatz zu anderen kabellosen Funktechnologien, die auf dem 2,4 GHz-Band basieren (WLAN, Bluetooth, ZigBee), weniger Interferenzen aufweist, da es im Frequenzbereich von unter einem Gigahertz agiert. Die maximale Übertragungsrate beträgt 100 kbit/s [34]. Die genauen Frequenzen dieses Standards betragen in Europa 868,42 MHz und in den USA 908,42 MHz. Z-Wave wurde in erster Linie für Smart Home Anwendungen im privaten und kommerziellen Bereich entwickelt und steht für einen geringen Energieverbrauch und hohe Kommunikationssicherheit. Die Bereitstellung dieses Protokolls und des dazugehörigen *SDKs (Software Development Kit)* geschieht unter einer Lizenz, die beim Hersteller Sigma Designs erworben werden kann. Eine Open-Source-Implementierung des Z-Wave-Protokolls, *open-zwave*, steht dennoch zur Verfügung, beinhaltet aber nicht die gleichen Sicherheitsmechanismen wie bei der kommerziellen Version [35].

### Mobilfunk

Das mobile Internet hat sich in den letzten Jahren bei den Download-Raten sehr fortschrittlich entwickelt (siehe Abbildung 6). Während mit der *2G-Technologie (Mobiles Netz der zweiten Generation, EDGE)* lediglich Daten-Download-Raten von etwa 236 kbit/s [36] erreicht werden konnte, bietet das heute verbreitete und von vielen Geräten unterstützte *Mobile Netz der vierten Generation (4G, LTE-Advanced, LTE-A)* Spitzenwerte einer Geschwindigkeit von bis zu einem Gigabit pro Sekunde. Diese erreichten Geschwindigkeiten stehen aber nicht am Ende der Entwicklung von Mobilfunkstandards. Denn es wird damit gerechnet, dass der nächste *5G-Standard* schon im Jahr 2020 Geschwindigkeiten bereitstellen könnte, die bis zu zehn Mal schneller sind und Latenzzeiten von unter einer Millisekunde aufweisen [37]. *LTE-A* würde sich aufgrund einer hohen Skalierbarkeit und einer langen Lebensdauer für Lösungen bei Smart Cities anbieten. In Netzwerken, wo es aber auf besonders energiesparende Komponenten ankommt, wäre der Mobilfunk als Übertragungsmedium aufgrund von erhöhtem Energieverbrauch nicht geeignet [5].

Die Entwicklung von Mobilfunk-Technologien werden von einem international agierenden Gremium namens *3rd Generations Partnership Project (3GPP)* vorangetrieben. Das 3GPP entwickelt momentan einen Standard, der für IoT von Bedeutung gewinnen könnte. *NarrowBand-IoT (LTE-Cat-NB1)* agiert im Mobilfunknetz LTE-A. Der Energiebedarf wird zugunsten von IoT-Anforderungen reduziert, indem die maximale Datenrate ebenfalls heruntergesetzt wird. Die gute Netzabdeckung aus dem LTE-Netz macht sich diese Technologie zum Vorteil und ist somit innerhalb als auch außerhalb von Gebäuden verfügbar [38].

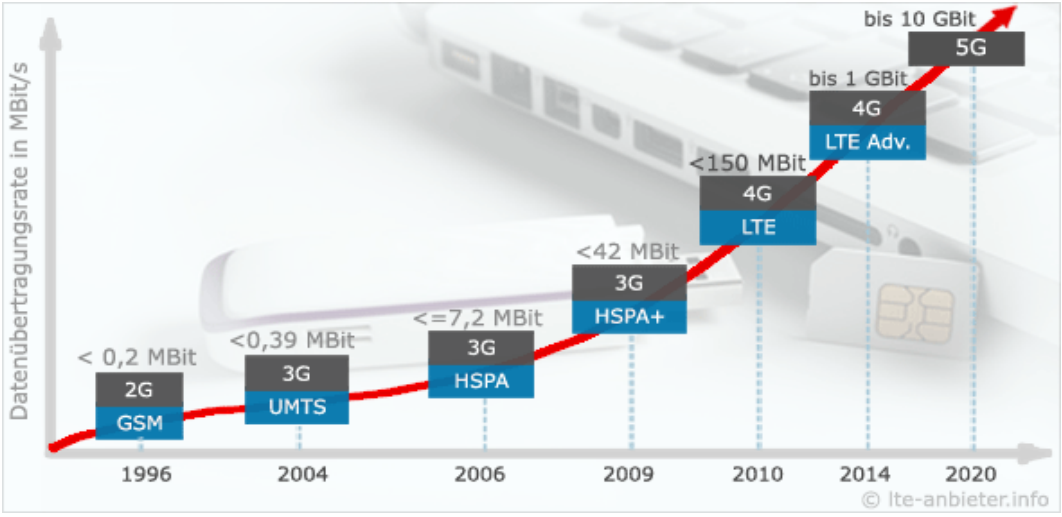



*Abbildung 6: Datenübertragungsraten der unterschiedlichen Mobilfunknetze [37]*

### Low Power Wide Area Networks

Für die Kommunikation über die Grenzen lokaler Netzwerke hinaus im Niedrig-Energie-Bereich werden sogenannte *Low Power Wide Area Networks (LPWANs)* verwendet. Diese haben in Relation zu den gängigen Funk-Techniken zwar einen geringeren Datendurchsatz, sind im Gegensatz zu Mobilfunk jedoch energiesparender und haben gegenüber WLAN oder Bluetooth eine größere Reichweite. So können Devices mit einer

Batterie in einem derartigen Netzwerk über mehrere Jahre betrieben werden. Entwickelt wurde LPWAN dafür, Daten von Sensoren über eine große Entfernung kostengünstig zu übertragen. Ein Netzwerk besteht bei dieser Technologie aus sogenannten *End Nodes* und *Gateways*. Die Frequenzen, über die die Geräte miteinander kommunizieren, sind die des ISM-Bands oder die von Mobilfunk. Die Gateways senden im nächsten Schritt die Daten über eine IP-Verbindung oder Mobilfunk an Netzwerkserver weiter (siehe Abbildung 7) [39]. Bekannte Vertreter von Standards von LPWANs sind beispielsweise *LoRaWAN, Weightless* oder *NarrowBand IoT (NB-IoT).*

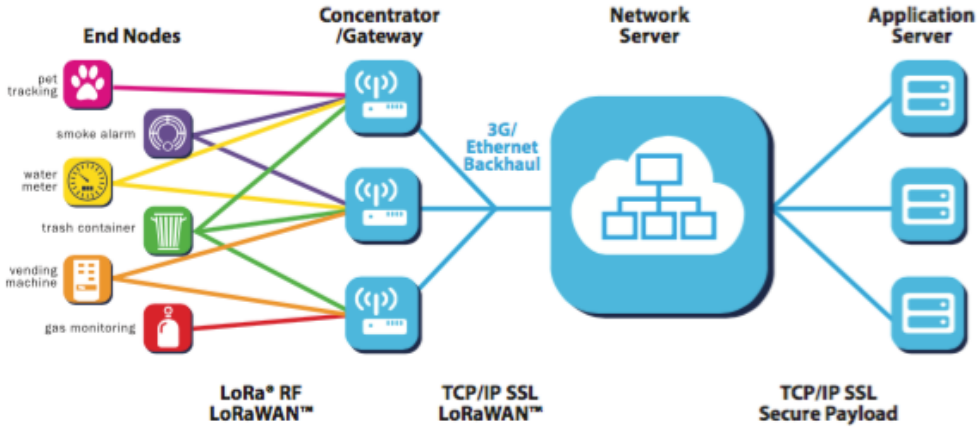



*Abbildung 7: Netzwerktopologie eines LPWAN bei LoRaWAN [39]*

## 2.6. IoT-Standards des Application Layer

*Anwendungsprotokolle (engl.: Application Protocols)* sind Protokolle, die im OSI-Referenzmodell in der obersten Schicht verwendet werden (siehe Abbildung 2, Seite 6). Das Hauptaugenmerk liegt darauf, Nachrichten zwischen Schnittstellen der Anwendungen zu übermitteln. Bekannte Vertreter von Anwendungsprotokollen im Internet sind etwa das Hypertext Transfer Protocol oder das File Transfer Protocol. Nun gibt es für den Bereich IoT spezielle Anforderungen, die ein IoT-Anwendungsprotokoll mitbringen sollte. Ein *geringer Overhead* wird mit der Zunahme von Devices im Internet ein wichtiger Aspekt sein, der unbedingt berücksichtigt werden muss. Die Netz-Bandbreite ist eine begrenzte Ressource und auch Rechner und Mikrokontroller, die beispielsweise batteriebetrieben sind, profitieren davon, wenn durch geringen Overhead ein geringerer Stromverbrauch entsteht. Die *Things* im Internet der Dinge weisen meist auch geringen Speicherplatz und wenig Prozessorleistung auf. *Leichtgewichtige Nachrichtenübermittlung* ist hier sehr bedeutsam und bewirkt, dass auch kleinste Geräte in ein IoT-System mit einbezogen werden können. Die *Unterstützung von Plattformen, Geräten* und *verschiedenen Datentypen* spielt auch eine wesentliche Rolle, da immer mehr Hersteller Lösungen anbieten, die auf unterschiedlichste Systeme mit verschiedenen Programmiersprachen basieren. Eine *hohe Zuverlässigkeit der Übertragung* muss in einem solchen IoT-Netzwerk dafür garantieren können, dass gerade bei Netzwerken mit schwankender Übertragungsqualität Nachrichten beim Empfänger ankommen. Die Verbindung der Kommunikationspartner muss *zuverlässig überwacht* werden, denn bei einer hohen Anzahl an Geräten in einem Netzwerk kann es häufig zum Ausfall von Hardware kommen. Die Unterstützung einer *flexiblen IoT-Architektur* ist wünschenswert,

denn die einfache intelligente Kopplung von heterogenen Objekten und Knoten innerhalb von IoT-Netzwerken wird zukünftig eine große Rolle spielen [40].

Die gängigsten IoT-Anwendungsprotokolle sind *CoAP (Constrained Application Protocol), MQTT (Message Queue Telementry Transport), XMPP (Extensible Messaging and Presence Protocol) und HTTP/1.1* und *HTTP/2*.

### XMPP

XMPP beschreibt einen auf XML basierenden offenen Kommunikationsstandard, der von der IETF unter den Namen *RFC 6120, 6121* und *6122* festgelegt wurde. Diese Spezifikation wurde 1999 von der *Jabber Open Source Community* entwickelt und diente ursprünglich dem Nachrichtenaustausch bei *Instant Messaging* Anwendungen *(IM)*. Es wird neben der Nachrichtenübermittlung als offenes, sicheres und dezentralisiertes Nachrichtenprotokoll für Präsenzerkennung, Multi-User-Chatting, Video- und Sprachanrufe und dem generellen Austausch von Daten in XML-Form verwendet [41]. Es unterstützt die Kompatibilität zu anderen Protokollen, Zugriffskontrolle, Ende-zu-Ende-Verschlüsselung und wird in kleinem Umfang in IoT eingesetzt [5]. Die Standard-Architektur dieses Kommunikationsprotokolls ist das Request-Response-Verfahren. Durch eine weitere Spezifikation *(XEP-0060)* kann XMPP um die Funktionalität des Publish/Subscribe-Pattern (siehe 3.3.1, Seite 21) erweitert werden [42]. Diese Eigenschaft qualifiziert dieses Protokoll durchaus für den Einsatz in IoT, doch in seiner Ursprungsform ist es nicht für limitierte umfassende Netzwerke geeignet, da es aufgrund hohem Overhead zu einer erhöhten Nutzung der Bandbreite und zu größerem Energie-Aufwand kommt [43].

### HTTP

HTTP ist ein zustandsloses Übertragungsprotokoll auf der Anwendungsschicht. Als zuverlässiges Transportprotokoll nutzt HTTP TCP. Der Zweck dieses Protokolls ist hauptsächlich Daten für Webseiten im Internet zu übertragen. Dafür sendet der Client einen *GET,*- oder *POST-Request,* der im Client-Server-Modell vom Server mit der *Response* beantwortet wird. Zum Übertragen einer Information ist zuvor ein Verbindungsaufbau nötig. Ab der Version HTTP/1.1 wird die Verbindung nicht mehr nach jeder Datenübermittlung geschlossen, sondern es können mehrere Requests hintereinander stattfinden [22]. Die seit 1999 existierende HTTP/1.1 Spezifikation ist unter dem *RFC 2616* standardisiert. Eine neuere Spezifikation, HTTP/2, soll das inzwischen in die Jahre gekommene HTTP/1.1 unter dem Standard *RFC 7540* ablösen. Dank Multiplexing (mehrere Anfragen werden zeitgleich zum Server geschickt) soll es Performance-Probleme aufgrund von ständig wachsendem Datenaufkommen beheben und Netzwerkressourcen besser nutzen, indem der Overhead reduziert wird und Kompression des Headers vorgenommen wird [44].

### CoAP

CoAP definiert ein Anwendungsprotokoll für IoT, das auf dem *REST (REpresentational State Transfer)* Modell basiert und unter *RFC 7252* standardisiert ist. Es benutzt *UDP* als Transportprotokoll (siehe Abbildung 2, Seite 6). Alternativ kann es auch mit TCP genutzt werden und nutzt wie HTTP Methoden wie *GET, POST, PUT* und *DELETE*. Verwendung findet CoAP in eingeschränkten Netzwerken, zum Beispiel in Low Power Wireless Personal Area Networks, steht für geringen Overhead und ist für M2M-Kommunikation konzipiert worden. Dank der einfachen Unterstützung von HTTP ist CoAP auch als Web

Transfer Protokoll bekannt. Die Daten werden hiermit über einen einfachen Weg zwischen Clients und Servern per Request/Response übermittelt [5].

### MQTT

Im Gegensatz zu CoAP benutzt MQTT als Protokoll der Transportschicht standardmäßig TCP. Die Entwickler von MQTT sprechen diesem Protokoll folgende Eigenschaften zu: Es soll ein leichtgewichtiges, effizientes, sicheres, ereignis- und nachrichtenorientiertes Anwendungsprotokoll sein. Zum Beispiel sei es für Verbindungen an abgelegenen Orten geeignet, wo Netzwerk-Datenübertragungsraten stark limitiert sind [8] [40]. Diesen Angaben folgend ist anzunehmen, dass die Spezifikationen von MQTT besonders auf die Anforderungen eines IoT-Anwendungsprotokolls passen. Diese Annahme soll in den kommenden Kapiteln untersucht werden, indem der Status Quo des Protokolls untersucht und dieses anhand eines Testbettes implementiert wird.

# 3. MQTT

## 3.1. Definitionen

### 3.1.1. Was ist MQTT?

Das *Message Queue Telemetry Transport* Protokoll, kurz *MQTT* (früher *MQ Telemetry Transport)*, ist ein ursprünglich von IBM entwickeltes leichtgewichtiges Kommunikationsprotokoll für M2M-Kommunikation im Internet of Things auf Anwendungsebene. Es wurde dafür entwickelt, an Orten mit limitierten Infrastrukturnetzen und energiesparenden Geräten mit einer begrenzten Leistung eine zuverlässige Nachrichtenvermittlung zu ermöglichen [45]. Aufgrund einer geringen Komplexität, einer hohen Flexibilität, geringen Overheads von MQTT-Paketen und der effizienten Verteilung von Nachrichten an einen oder mehrere Empfänger ist MQTT ebenfalls für eingebettete Systeme und bei der Kommunikation zwischen Netzwerken, Anwendungen und Middleware geeignet [6]. Verwendung findet das Protokoll unter anderem auch beim Zustellen von Push-Nachrichten beim *Facebook Messenger* und bei Diensten von *Amazon Web Services IoT* [9] [46].

Mittlerweile ist MQTT ein Open-Source-Protokoll, das seit 2013 von der *Organization for the Advancement of Structured Information Standards (OASIS)* standardisiert wird [5]. Die aktuelle Version von MQTT ist die MQTT v3.1.1 Spezifikation [45]. Eine weitere Spezifikation von MQTT ist *MQTT-SN*. Diese ist sehr ähnlich zu MQTT, spezialisiert sich aber auf Sensornetzwerke (WSNs). MQTT-SN ist eine eingeschränkte Version, zugeschnitten auf die Implementierung von batteriebetriebenen Geräten, die gegen Übertragungsfehler geschützt werden sollen und mit einer extrem niedrigen Übertragungsrate und noch kleineren Datenpaketen agieren. Ursprünglich wurde MQTT-SN dazu entwickelt, als Anwendungsprotokoll auf dem Stack von ZigBee zu laufen [47].

### 3.1.2. Telemetrie-Daten

Telemetrie beschreibt die Übermittlung von Messdaten von einer bestimmten Stelle, beispielsweise von einem Sensor, zu einem festgelegten Ziel, einer zentralen Anwendung oder einem Endknoten. Gemessene Umgebungsbedingungen und Geräteparameter können am Ziel gesammelt und ausgewertet werden. Die zu übertragenen Daten werden auch Telemetrie-Daten genannt. Telemetrie-Daten fallen unter anderem bei Wetter-Messungen, bei Datenerhebungen aus Fahrzeug,- Flug- und Raumtechnik oder bei der Übertragung von Medizinischen Daten an. Auch in IoT-Systemen wie in einer Smart City fallen überdurchschnittlich viele Messdaten an, die übermittelt werden müssen. Eine gemeinsame Eigenschaft dieser Telemetrie-Daten ist die geringe Größe von Messwerten. MQTT geht auf diese Anforderung ein, hiermit lassen sich in erster Linie Nachrichten mit kleinem Inhalt auch über limitierte Netze übertragen [48] [49].

### 3.1.3. Machine-to-Machine-Kommunikation

Der Begriff *Machine-to-Machine-Kommunikation*, auch *M2M-Kommunikation* genannt, ist sehr weit gefasst. Es ist nicht ein einzelner Kommunikationsschritt, sondern kann ganze Prozesse umfassen. M2M steht dafür, dass Geräte (bidirektional) Informationen

mit Anwendungen anderer Devices automatisch über ein Kommunikationsnetzwerk austauschen können. Deswegen kann auch von *M2(CN2)M* gesprochen werden, von *Machine-to-(Communication-Network-to-)Machine Kommunikation.*

Der Datenfluss bei M2M-Kommunikation kann sehr unterschiedlich sein. Es können nicht nur einzelne Geräte mit einer Anwendung kommunizieren, sondern Gruppen von Geräten über Gateways in das Internet gelangen. In einem IoT-Netzwerk muss die Kommunikation nicht zwingend von einer Maschine ausgehen. Befehle werden unter anderem auch von Menschen getriggert. Eine Eingrenzung des Begriffs M2M wird in diesem Fall bei *dem* Gerät vorgenommen, bei dem die Nutzereingabe geschieht. Hier sitzt der sogenannte *End-Point* von M2M-Beziehungen [49] [50].

## 3.2. Entstehung von MQTT

Wie schon angedeutet, war die Intention hinter der Entwicklung von MQTT, ein extrem leichtgewichtiges ressourcenschonendes Kommunikationsprotokoll zu entwerfen. MQTT wurde bereits im Jahr 1999 von *IBM*-Mitarbeiter Andy Stanford-Clark und Arlen Nipper *(Arcom*, jetzt *Cirrus Link)* erfunden, um zu der damaligen Zeit Öl-Pipelines über eine Satelliten-Verbindung miteinander zu vernetzen. Der Einsatzzweck des Protokolls hat sich mit der Zeit von kommerziellen eingebetteten Systemen zu offenen IoT-Anwendungen geändert, was unter anderem auch daran lag, dass *MQTT v3.1* im Jahr 2010 zur freien Verfügung gestellt wurde. Das Akronym „MQ Telemetry Transport" bezieht sich auf ein von IBM entwickeltes plattformunabhängiges nachrichtenorientiertes Middleware-Programm namens *MQSeries (jetzt*: *IBM MQ)*, das 1992 entwickelt wurde und sich mit *Message Queueing* beschäftigte. Die heutige Abkürzung MQTT ist jedoch die korrekte offizielle Bezeichnung [51].

Mit der Veröffentlichung dieses Kommunikationsprotokolls wurde es ermöglicht, dass Implementierungen wie *HiveMQ* oder das *Paho Project* der *Eclipse Foundation* diesem zu einer erhöhten Popularität verhalfen, was in 2013 in der Festlegung eines Standards gipfelte. OASIS, eine Vereinigung zur Festlegung von Standards, veröffentlichte am 29. Oktober 2014 *MQTT v3.1* als offiziellen OASIS-Standard. Seit 2016 ist Message Queue Telemetry *Transport* in der *Version 3.1.1* auch ein ISO-Standard mit der Bezeichnung *ISO/IEC 20922:2016* [52]. Die *Version 5* des Protokolls ist in einem Paper von OASIS im Mai 2017 als *Working Draft* gekennzeichnet worden, erste Implementierungen sind momentan in Entwicklung [53]. Sämtliche folgenden Ausführungen zum Aufbau und zur Funktionsweise von MQTT beziehen sich auf die Version MQTT v3.1.1. Dabei werden als Quellen die Spezifikation von OASIS [54], der Blog von *HiveMQ* zu den MQTT Grundlagen [55] sowie *Das MQTT-Praxisbuch* von Walter Trojan [40] verwendet.

## 3.3. Funktionsweise und Aufbau des Protokolls

### 3.3.1. Publish/Subscribe-Architektur

Das *Publish/Subscribe-Modell* beschreibt die Kommunikationsarchitektur von MQTT und steht hinter den Anforderungen von MQTT, ein skalierbares System mit einer hohen Flexibilität und einer geringen Komplexität zu sein. Das Schema dieser Architektur ist in Abbildung 8 veranschaulicht. Das Publish/Subscribe-Verfahren stellt sich als Alternative zum traditionellen *Client-Server-Modell* dar, bei dem der Server auf einen *Request* vom Client mit einer *Response* antwortet [56].

**Broker**
Bei diesem ereignisgesteuerten Ansatz gibt es eine Hauptkomponente, die zum einen die globalen Daten und zum anderen die Verbindungen von der zweiten Komponente, den *Clients*, verwaltet. Der Server ist somit für den Nachrichtenaustausch und für die Filterung von Nachrichten verantwortlich. In der Implementierung dieses Schemas nennt sich die zentrale Stelle *Broker*. Dieser Broker ist ein zentralisierter Server, der Daten zwischen den verschiedenen *Clients* vermittelt, wenn sich diese mit dem Broker verbunden haben [56] [48].

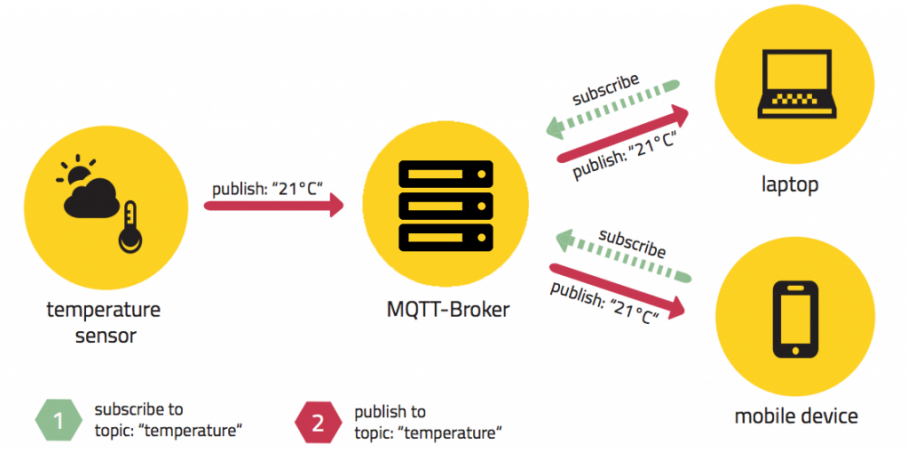



*Abbildung 8: Publish/Subscribe-Modell [40]*

**Client**
Beim Publish-Subscribe-Pattern sind zwei Arten von Clients vorhanden. Der Client, der Daten an den Broker sendet, nennt sich *Publisher (Producer)*. Ein Client, der sich beim Broker subskribiert und von diesem Nachrichten erhalten will, heißt *Subscriber (Consumer)*.

**Ablauf der Kommunikation**
Der Kommunikationsablauf stellt sich wie folgt dar: Wenn der Broker eine Nachricht vom Publisher bekommt, liefert er die Nachricht an subskribierende Clients aus. Es ist auf der einen Seite möglich, dass mehrere Clients gleichzeitig publishen, auf der anderen Seite kann der Broker auch eine Nachricht an mehrere Subscriber zur selben Zeit broadcasten. Ein Client, im Folgenden auch Knoten genannt, kann entweder ein Publisher, ein Subscriber, oder zur gleichen Zeit beides sein [57] [48].

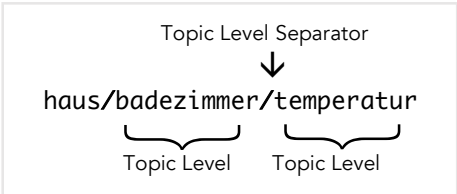
#### Vorteile des Verfahrens

Das Publish/Subscribe-Prinzip hat gegenüber dem Request/Response-Verfahren den entscheidenden Vorteil, dass bei einem großen System mit vielen Clients diese durch die Mehrfach-Beziehungen des Brokers gleichzeitig mit Nachrichten versorgt werden können. In einem IoT-System mit mehreren Sensoren und/oder vielen Aktoren, wie beispielsweise in einem WSN, eignet sich dieses Prinzip genauso wie bei Anwendungen, wo Nachrichten in unregelmäßigen Abständen eventgesteuert verschickt werden [48].

Ein weiterer vorteiliger Aspekt ist, dass für eine Kommunikation keine Ports oder IP-Adressen der Empfänger bekannt sein müssen. Publisher und Subscriber müssen folglich nichts von der Existenz voneinander wissen. Zusätzlich müssen diese auch nicht zur gleichen Zeit ein aktiver Bestandteil des Systems sein, denn der Broker speichert die Werte in einer Session, sodass ein hinzukommendes Objekt in diesem System direkt den zuletzt gespeicherten Wert zugeschickt bekommt [40].

### 3.3.2. Adressierung – Topic Names

Der Nachrichtenaustausch zwischen Client und Broker, beziehungsweise dessen Adressierung wird über sogenannte *Topics* gesteuert. Topics bestehen technisch aus einem UTF-8-String, stellen den Betreff der jeweiligen Nachricht dar und sind logisch ähnlich wie eine URL aufgebaut. Ein Beispiel für ein Topic wäre:




An dieser Stelle würde ein im Badezimmer befindlicher Client mit einem Sensor für Temperatur unter dem genannten Topic Werte an den Broker publizieren. Wenn die Nachricht unter diesem Topic beim Broker eingetroffen ist, wird sie von diesem unter dem gleichen Topic an subskribierende Clients weitergeleitet [40]. Der exakte Aufbau eines Datenpakets, das beim Publishen versendet wird, inklusive des *Payloads*, also dem eigentlichen Inhalt der Nachricht, kann in Abschnitt 3.3.4 eingesehen werden.

Ein Topic kann eine beliebige Anzahl an sogenannten *Levels* haben. Der *Topic Level Separator*, als Zeichen wird hier der *Slash ('/')* verwendet, separiert mehrere Level innerhalb eines hierarchischen Baums (siehe Abbildung 9, Seite 23). Ein Subscriber hat die Möglichkeit, mehr als ein Topic zu abonnieren [58].

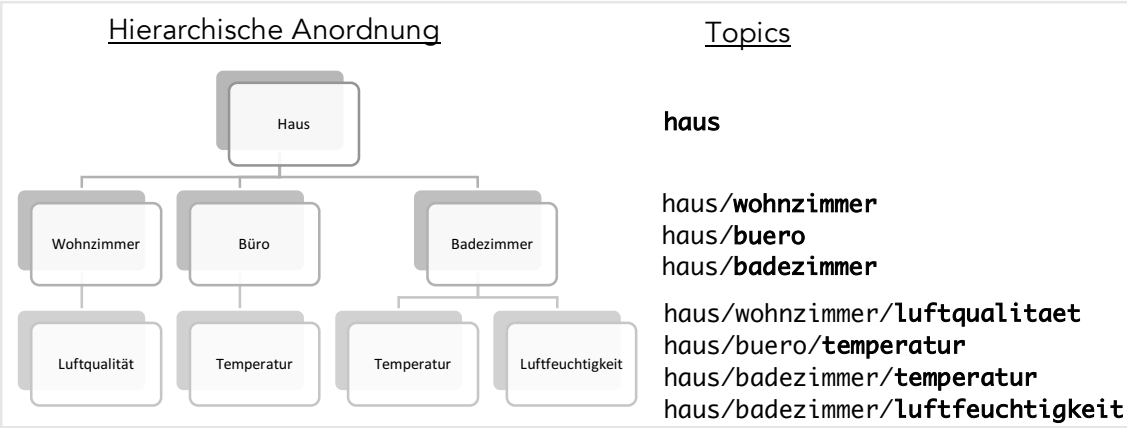


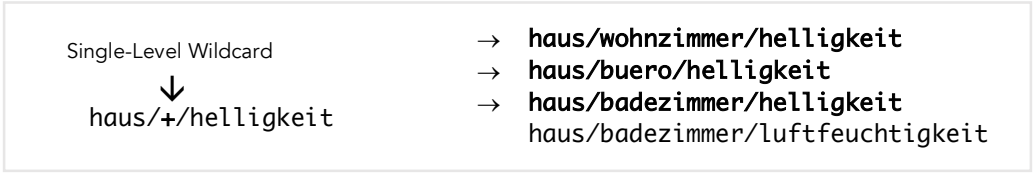

*Abbildung 9: Hierarchische Struktur von Topics*

Am Anfang und Ende eines Topics steht in der Regel kein Level Separator. Wenn am Anfang ein Separator steht, sorgt das für ein weiteres Topic Level. Eine Voraussetzung für einen validen Topic Name ist, dass dieser aus mindestens einem Zeichen besteht. Auch ein einzeln stehender Slash ist ein gültiges Topic. Zusätzlich findet eine Unterscheidung zwischen Groß- und Kleinschreibung statt. Auch Leerzeichen innerhalb eines Topics werden berücksichtigt und vom Broker akzeptiert, darauf sollte aber aufgrund von Lesbarkeit verzichtet werden [40] [54].

### 3.3.3. Adressierung – Filterung

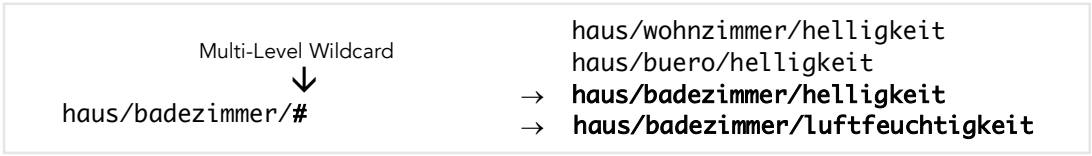
Neben der Filterung durch die Topic Levels wird mithilfe von *Topic Wildcards* eine Möglichkeit geboten, spezielle Topics bei einer Subskription herauszufiltern. Bei einem Vorgang, wo ein Client published, wird keine Möglichkeit einer Filterung angeboten. Wildcards ermöglichen es Subscribern, durch das Erstellen von Topic Filtern nur Topics zu abonnieren, die auf den Filter passen [58].

*Single-Level Wildcard +*
Diese Wildcard wählt Nachrichten des Levels aus, wo der Platzhalter steht. Als Platzhalter für diese Art von Filterung wird ein *Plus ('+')* verwendet. In folgendem Beispiel würde die Helligkeit in allen Zimmern im Haus abgefragt werden. Die betroffenen Topics sind mit einem Pfeil versehen. Auf der linken Seite steht die entsprechende Single-Level Wildcard. Eine Mehrfachverwendung dieser Wildcard ist möglich. Sie kann in jedem Level des Topics als Filter eingesetzt werden. Auch kann sie gänzlich allein als Topic Filter stehen [59].




Ein invalides Filter Topic wäre beispielsweise: haus+

*Multi-Level Wildcard #*
Die Multi-Level Wildcard repräsentiert das Level, wo sie eingesetzt wird und alle Darauffolgenden. Das ist auch der Grund, warum das hier verwendete *Hashtag-Zeichen ('#')* immer am Ende eines Topics stehen muss. Es werden in diesem Fall alle Topics abonniert, die sich auf das Badezimmer beziehen [59].




Ein invalides Filter Topic wäre: `haus/badezimmer#` oder `haus/#/helligkeit`

*Topics mit $*
Die Benennung von Topics kann sehr frei vorgenommen werden. Topics, die jedoch mit einem *Dollar-Zeichen ('$')* beginnen, werden gesondert behandelt. Diese sind für interne Server-Daten reserviert und nicht für den Informationsaustausch zwischen Clients bestimmt. Deshalb wird ein Publishen unter einem Topic mit diesem Zeichen am Anfang vom Server nicht akzeptiert. In der Regel nutzt der Broker die Zeichenkette *('$SYS/')* für die Serverspezifischen Statusinformationen [54]. Einige Beispiele sind:

```
$SYS/broker/clients/connected
$SYS/broker/messages/received
$SYS/broker/uptime
$SYS/broker/version
```

### 3.3.4. Paketaufbau

Beim Austausch von Nachrichten über MQTT werden Datenpakete in definierter Richtung übermittelt. Jedes sogenannte *MQTT Control Packet* lässt sich in drei Bereiche aufteilen und ist folgendermaßen untergliedert:

| |
|---|
| *Fixed Header* (in allen Control Packets vorhanden) |
| *Variable Header* (in einigen Control Packets vorhanden) |
| *Payload* (in einigen Control Packets vorhanden) |

*Abbildung 10: Struktur eines MQTT Control Packets*

*Fixed Header*
Der Fixed Header ist wie folgt aufgebaut und wird in einem MQTT Control Packet zwingend benötigt:

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *MQTT Control Packet Type* | | | | *DUP Flag* | *QoS Level Flag* | | *RETAIN Flag* |
| Byte 2... | *Remaining Length* | | | | | | | |

*Abbildung 11: Format des Fixed Header*

Der in Byte 1 an der Stelle Bit vier bis sieben definierte MQTT Control Packet Type wird im Abschnitt *Nachrichtenarten (Abschnitt 3.3.5)* besprochen. Die Besetzung der dazu-

gehörigen Flags (Bit null bis drei) werden ebenfalls in den nächsten Abschnitten behandelt. Die entsprechende Definition der Remaining Length (im zweiten Byte enthalten) kann in der Spezifikation von MQTT nachgeschlagen werden [54]. Die Remaining Length ist die Länge des Variable Header plus der Länge des Payloads (siehe Abbildung 16, Seite 27).

*Variable Header*

Einige MQTT Control Packets beinhalten einen Variable Header. Der Inhalt des Variable Header ist von dem entsprechenden Packet Type abhängig. Er beinhaltet meist einen zwei Byte langen *Packet Identifier.* Eine Übersicht, bei welchem Control Packet ein Packet Identifier verwendet werden muss, wird in der Spezifikation gegeben [54]. Die Abkürzungen *MSB* und *LSB* stehen bei Integer Werten jeweils für das *Most Significant Byte* beziehungsweise das *Last Significant Byte*.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *Packet Identifier MSB* | | | | | | | |
| Byte 2... | *Packet Identifier LSB* | | | | | | | |

*Abbildung 12: Formate des Packet Identifiers*

*Payload*

Bei manchen Control Packets ist der Payload optional oder erforderlich. Ob ein Payload erforderlich bzw. optional ist, kann in Abbildung 16 auf Seite 27 eingesehen werden. Der Payload besteht beim Fall des Publishens aus der sogenannten *Application Message*. Die Application Message entspricht den tatsächlichen Daten einer Anwendung, die von MQTT über das Netzwerk übermittelt werden. Damit verbunden sind ebenfalls der Topic Name (siehe 3.3.2 Adressierung – Topic Names, Seite 22) und die Faktoren für *Quality of Service* (siehe 3.3.8 – QoS, Seite 34).

### 3.3.5. Nachrichtenarten

Von MQTT werden 14 unterschiedliche Kontrollpakete zwischen Server und Clients versendet. In folgender Tabelle sind sämtliche MQTT Control Packet Types mit ihren zugehörigen Werten aufgelistet.

| Name | Wert | Richtung | Beschreibung |
|---|---|---|---|
| Reserviert | 0 | Verboten | Reserviert |
| CONNECT | 1 | Client zum Server | Client Request, um mit Server zu verbinden |
| CONNACK | 2 | Server zum Client | Connect Bestätigung |
| PUBLISH | 3 | Client zum Server oder Server zum Client | Publish Nachricht |
| PUBACK | 4 | Client zum Server oder Server zum Client | Publish Bestätigung |
| PUBREC | 5 | Client zum Server oder Server zum Client | Publish empfangen (QoS 2, Teil 1) |
| PUBREL | 6 | Client zum Server oder Server zum Client | Publish freigegeben (QoS 2, Teil 2) |
| PUBCOMP | 7 | Client zum Server oder Server zum Client | Publish vollständig (QoS 2, Teil 3) |
| SUBSCRIBE | 8 | Client zum Server | Subscribe Anfrage |
| SUBACK | 9 | Server zum Client | Subscribe Bestätigung |
| UNSUBSCRIBE | 10 | Client zum Server | Unsubscribe Anfrage |
| UNSUBACK | 11 | Server zum Client | Unsubscribe Bestätigung |
| PINGREQ | 12 | Client zum Server | PING Anfrage |
| PINGRESP | 13 | Server zum Client | PING Anwort |
| DISCONNECT | 14 | Client zum Server | Client beendet Verbindung |
| Reserviert | 15 | Verboten | Reserviert |

*Abbildung 13: MQTT Control Packet Types*

| Control Packet | Packet Identifier Field | Payload |
|---|---|---|
| CONNECT | Nein | Erforderlich |
| CONNACK | Nein | Nein |
| PUBLISH | Ja, wenn QoS > 0 | Optional |
| PUBACK | Ja | Nein |
| PUBREC | Ja | Nein |
| PUBREL | Ja | Nein |
| PUBCOMP | Ja | Nein |
| SUBSCRIBE | Ja | Erforderlich |
| SUBACK | Ja | Erforderlich |
| UNSUBSCRIBE | Ja | Erforderlich |
| UNSUBACK | Ja | Nein |
| PINGREQ | Nein | Nein |
| PINGRESP | Nein | Nein |
| DISCONNECT | Nein | Nein |

*Abbildung 14: Control Packets – Zuordnung Packet Identifier und Payload*

| Control Packet | Fixed Header Flags | Bit 3 | Bit 2 | Bit 1 | Bit 0 |
|---|---|---|---|---|---|
| CONNECT | Reserviert | 0 | 0 | 0 | 0 |
| CONNACK | Reserviert | Nein | 0 | 0 | 0 |
| PUBLISH | In MQTT 3.3.1 | DUP | QoS | QoS | RETAIN |
| PUBACK | Reserviert | 0 | 0 | 0 | 0 |
| PUBREC | Reserviert | 0 | 0 | 0 | 0 |
| PUBREL | Reserviert | 0 | 0 | 1 | 0 |
| PUBCOMP | Reserviert | 0 | 0 | 0 | 0 |
| SUBSCRIBE | Reserviert | 0 | 0 | 1 | 0 |
| SUBACK | Reserviert | 0 | 0 | 0 | 0 |
| UNSUBSCRIBE | Reserviert | 0 | 0 | 1 | 0 |
| UNSUBACK | Reserviert | 0 | 0 | 0 | 0 |
| PINGREQ | Reserviert | 0 | 0 | 0 | 0 |
| PINGRESP | Reserviert | 0 | 0 | 0 | 0 |
| DISCONNECT | Reserviert | 0 | 0 | 0 | 0 |

*Abbildung 15: Fixed Header Flags*

| Stelle | Von | Bis |
|---|---|---|
| 1 | 0 (0x00) | 127 (0x7F) |
| 2 | 128 (0x80, 0x01) | 16 383 (0xFF, 0x7F) |
| 3 | 16 384 (0x80, 0x80, 0x01) | 2 097 151 (0xFF, 0xFF, 0x7F) |
| 4 | 2 097 152 (0x80, 0x80, 0x80, 0x01) | 268 435 455 (0xFF, 0xFF, 0xFF, 0x7F) |

*Abbildung 16: Größe des Remaining Length Field*

### 3.3.6. Verbindungsaufbau

**CONNECT**

Alle Clients müssen vor dem Nachrichtenaustausch eine Verbindung zum Broker aufbauen. Deswegen ist das erste Kontrollpaket, das vom Client zum Server geschickt wird, das *CONNECT Control Packet* (siehe Abbildung 17). Es wird nur *einmal* geschickt. Sollte das Paket ein zweites Mal beim Server eintreffen, wird eine bestehende Verbindung zum Client getrennt.

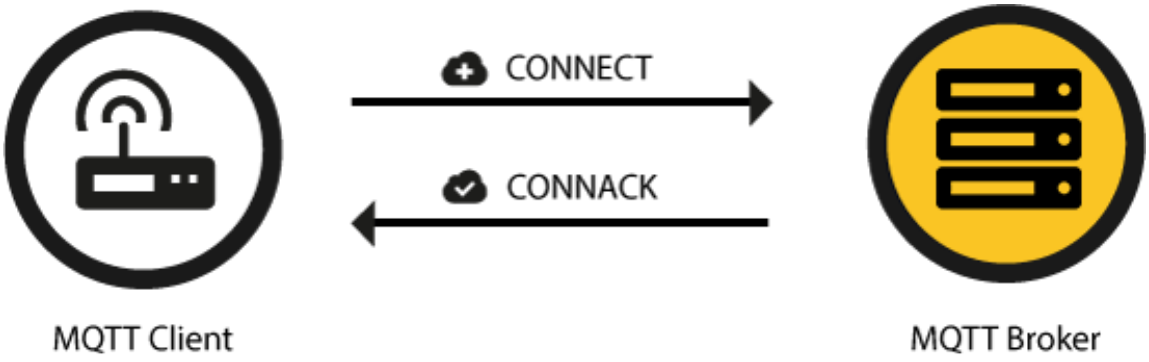



*Abbildung 17: MQTT Verbindungsaufbau über CONNECT UND CONNACK [60]*

Fixed Header:

Der Packet Type Wert ist an dieser Stelle *1*. Die Bits null bis drei sind für spätere Versionen des Protokolls reserviert, müssen aber mit den vorgegebenen Werten gefüllt werden, da das Paket sonst vom Server abgelehnt wird.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *MQTT Control Packet Type (1)* | | | | *Reserved* | | | |
| | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Byte 2... | *Remaining Length* | | | | | | | |

*Abbildung 18: CONNECT Fixed Header*

Variable Header:

Der Variable Header beinhaltet den *Protocol Name*, das *Protocol Level*, die *Connect Flags* und Werte für *Keep Alive*. Der Protocol Name ist ein UTF-8 String, der wie das Protokoll "MQTT" lautet. Der Name ist zur eindeutigen Erkennung als MQTT-Paket beispielsweise für Firewalls gedacht. Das Protocol Level beschreibt die Version des verwendeten Protokolls. Im Fall von MQTT 3.1.1 ist der Wert "4". Die genauen Erläuterungen zu den einzelnen Funktionalitäten der restlichen Flags werden in den folgenden Kapiteln gegeben.

| Bit | Beschreibung | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| *Protocol Name* | | | | | | | | | |
| Byte 1 | Länge MSB (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Byte 2 | Länge LSB (4) | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| Byte 3 | 'M' | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| Byte 4 | 'Q' | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| Byte 5 | 'T' | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| Byte 6 | 'T' | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| *Protocol Level* | | | | | | | | | |
| Byte 7 | Level (4) | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| *Connect Flags* | | | | | | | | |
| | User Name Flag | Password Flag | Will Retain | Will QoS | | Will Flag | Clean Session | Reserved |
| Byte 8 | X | X | X | X | X | X | X | 0 |
| *Keep Alive MSB* | | | | | | | | |
| Byte 9 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| *Keep Alive LSB* | | | | | | | | |
| Byte 10 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

*Abbildung 19: CONNECT Variable Header*

Payload:
Der Payload des CONNECT Packets ist abhängig von den im Variable Header definierten Feldern. Wenn diese Felder aktiviert wurden, ist die Reihenfolge der hier erforderlichen Felder *Client Identifier (ClientId), Will Topic, Will Message, Username* und *Password*. Der Client Identifier identifiziert jeden MQTT Client eindeutig, der sich mit dem Broker verbindet. Die ClientId ist pro Server einzigartig und hilft dabei, den aktuellen Status des Clients zu identifizieren. Das Will Topic wird benötigt, wenn das Will Flag auf *1* gesetzt wird (siehe 3.3.12 Weitere Features, Seite 47).

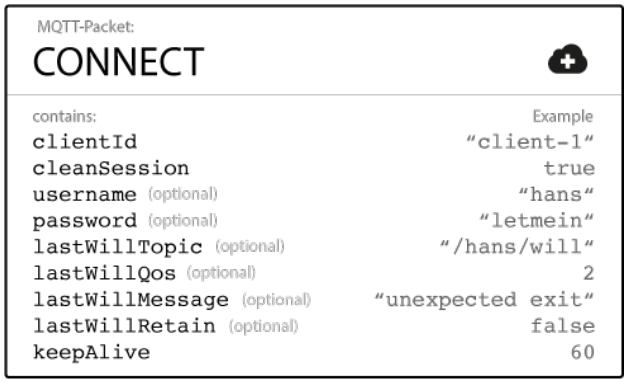



*Abbildung 20: Beispiel einer CONNECT Message [60]*

### CONNACK

Nach der Initiierung des Verbindungsaufbaus durch den Client (CONNECT) antwortet der Server mit der *CONNACK* Nachricht (siehe Abbildung 17, Seite 28). Dies ist das erste Control Packet vom Server an den Client. Wenn der Client das Paket nicht empfängt, schließt dieser die Verbindung nach einem Timeout. Wenn der Server einen *Return Code* mit dem Wert *0* schickt, ist die Verbindung hergestellt.

Fixed Header:
Der Control Packet Type lautet hier CONNACK. Die Bits null bis drei des ersten Byte sind reserviert und die Remaining Length kann in Abbildung 16 auf Seite 27 nachgeschlagen werden.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *MQTT Control Packet Type (2)* | | | | *Reserved* | | | |
| | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| Byte 2... | *Remaining Length (2)* | | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

*Abbildung 21: CONNACK Fixed Header*

Variable Header:
Das erste Bit vom ersten Byte ist das *Session Present (SP) Flag.* Die restlichen Bits sind reserviert und müssen null sein. Die Bedeutung des *Connect Return Code* kann in Abbildung 23 auf Seite 31 nachgeschlagen werden.

| Bit | Beschreibung | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| *Connect Acknowledge Flags* | | *Reserved* | | | | | | | *SP* |
| Byte 1 | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | X |
| *Connect Return Code* | | | | | | | | | |
| Byte 2 | | X | X | X | X | X | X | X | X |

*Abbildung 22: CONNACK Variable Header*

*Session Present Flag:* Wenn beim Verbinden (CONNECT) das Flag Clean Session aktiviert wurde (1), muss der Server beim CONNACK Packet Session Present auf 0 setzen. Das Session Present Flag ermöglicht es, dem Client zu erfahren, ob der Server schon zuvor eine vorhandene Session State gespeichert hat (siehe 3.3.12 Weitere Features, Seite 47).

| Wert | Return Code Antwort | Beschreibung |
|---|---|---|
| 0 | 0x00 Connection Accepted | Die Verbindung wurde akzeptiert |
| 1 | 0x01 Connection Refused, unacceptable protocol version | Der Server unterstützt das Level (Version) des vom Client verwendeten MQTT Protokolls nicht |
| 2 | 0x02 Connection Refused, identifier rejected | Der Client Identifier ist korrekt formatiert, aber wird nicht vom Server angenommen |
| 3 | 0x03 Connection Refused, Server unavailable | Die Netzwerkverbindung wurde hergestellt, der MQTT Service ist aber nicht verfügbar |
| 4 | 0x04 Connection Refused, bad name or password | Die Daten im Nutzernamen oder Passwort sind nicht in der richtigen Form |
| 5 | 0x05 Connection Refused, not authorized | Der Client ist nicht für eine Verbindung autorisiert |
| 6-255 | | Reserviert für zukünftige Versionen |

*Abbildung 23: CONNACK Connect Return Code*

Payload:
Die CONNACK Nachricht hat keinen Payload (siehe Abbildung 14, Seite 26).

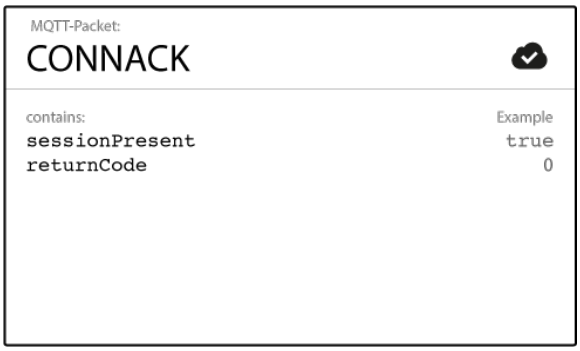



*Abbildung 24: Beispiel einer CONNACK Message [60]*

### 3.3.7. Publizieren einer Nachricht

**PUBLISH**

Nachdem die Verbindung zum Server erfolgreich hergestellt wurde, kann der Client die *PUBLISH* Message dafür benutzen, eine *Application Message* an den Server zu senden. Die zweite Möglichkeit ist, dass eine Application Message vom Server zu Clients distribuiert wird, die zuvor eine Subskription zu dem betreffenden Topic durchgeführt haben. An dieser Stelle findet der Austausch der eigentlichen Information statt.

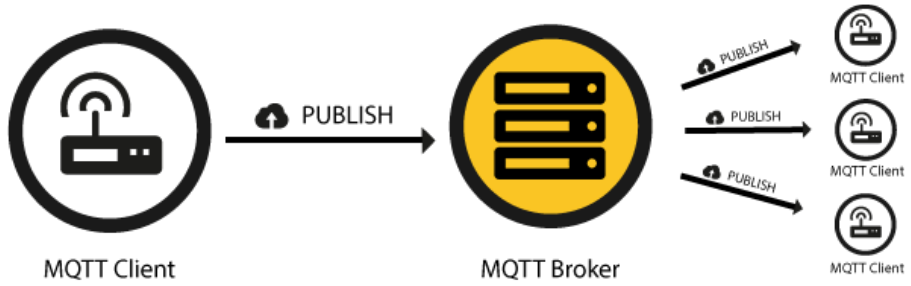



*Abbildung 25: Publishing einer Nachricht [60]*

Fixed Header:
Der Packet Type PUBLISH wird im Fixed Header festgelegt (3). Das Setzen, beziehungsweise die Bedeutung der Flags der Bits null bis drei werden im Folgenden erläutert.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *MQTT Control Packet Type (3)* | | | | *DUP Flag* | *QoS Level Flag* | | *RETAIN Flag* |
| | 0 | 0 | 1 | 1 | X | X | X | X |
| Byte 2 | *Remaining Length* | | | | | | | |

*Abbildung 26: PUBLISH Fixed Header*

- *DUP Flag*: Wenn dieses Flag ist auf 1 gesetzt ist, bedeutet das, dass die betroffene Nachricht ein Duplikat einer bereits gesendeten, nicht quittierten Nachricht ist. Wenn der Wert 0 ist, ist es der erste Versuch, diese Nachricht zu übertragen. Das DUP Flag ist nur relevant bei Nachrichten mit einem QoS Level größer als 0 (siehe 3.3.8 – QoS, Seite 34).

- *QoS Flags:* Diese Flags repräsentieren das Quality of Service Level einer Nachricht. Je nach Level (0,1 oder 2) kann eine Garantie gegeben werden, wie zuverlässig eine Nachricht ankommt (siehe 3.3.8 – QoS, Seite 34). Dies geschieht mit einem Response Control Packet, was sich je nach QoS Level unterscheidet (siehe Abbildung 27).

| QoS Level | Erwartete Antwort |
|---|---|
| QoS 0 | keine |
| QoS 1 | PUBACK Packet |
| PUBLISH | PUBREC Packet |

*Abbildung 27: Erwartete PUBLISH Packet Antwort*

- *RETAIN Flag:* Retained Nachrichten geben an, dass die betreffende Nachricht vom Server gespeichert werden soll, damit neu hinzukommende Clients diese Nachricht unter dem angegebenen Topic direkt empfangen können (siehe 3.3.12 Weitere Features, Seite 47).

Variable Header:
Der *Topic Name* entspricht einem UTF-8 String und steht im Variable Header an erster Stelle. Eine genaue Beschreibung des Topic Name wird im vorigen Abschnitt *Adressierung* gegeben (siehe 3.3.2 Adressierung – Topic Names, Seite 22). In diesem Beispiel lautet der Topic Name `topic/1`. Das Feld *Packet Identifier* ist nur in PUBLISH Packets enthalten, die ein höheres QoS Level als 0 haben.

| Bit | Beschreibung | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| *Topic Name* | | | | | | | | | |
| Byte 1 | Länge MSB (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Byte 2 | Länge LSB (7) | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| Byte 3 | 't' (0x74) | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| Byte 4 | 'o' (0x6f) | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| Byte 5 | 'p' (0x70) | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| Byte 6 | 'i' (0x69) | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| Byte 7 | 'c' (0x63) | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| Byte 8 | '/' (0x2f) | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| Byte 9 | '1' (0x31) | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| *Packet Identifier* | | | | | | | | | |
| Byte 10 | Packet Identifier MSB (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Byte 11 | Packet Identifier LSB (10) | 0 | 0 | 0 | 0 | X | 0 | X | 0 |

*Abbildung 28: PUBLISH Variable Header Beispiel*

Payload:
Jede PUBLISH Nachricht hat in der Regel einen Payload, der die eigentliche Application Message beinhaltet, also die eigentlich zu übertragenden Daten in einem Byte-Datenstrom. Bei dem Inhalt und Format der Daten kommt es auf die Anwendung an. Mit MQTT ist es möglich, Binär-Daten oder textbasierte Daten beispielsweise im XML- oder JSON-Format zu versenden.

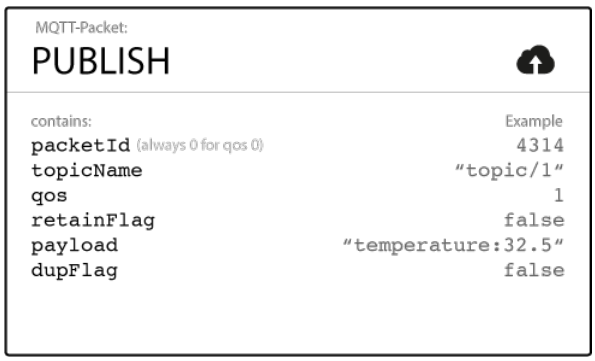



*Abbildung 29: Beispiel einer PUBLISH Message [60]*

### 3.3.8. Dienstgüte - Quality of Service (QoS)

*Definition: Quality of Service*, kurz *QoS* ist die Dienstgüte eines Kommunikationskanals. Das QoS Level ist die Vereinbarung von Sender und Empfänger, welche Garantie gegeben wird, damit eine Nachricht verbindlich beim Empfänger ankommt. Quality of Service ist ein wichtiger Aspekt von MQTT. Gerade bei limitierten Netzwerken, die eine unzuverlässige Datenübertragung bieten, kann dank der QoS Level die gewünschte Zuverlässigkeit gewählt werden. Bei MQTT gibt es drei verschiedene QoS Level, in denen die Nachrichten zugestellt werden sollen:

- Höchstens einmal (QoS Level 0)
- Mindestens einmal (QoS Level 1)
- Genau einmal (QoS Level 2)

Die Anwendung der unterschiedlichen QoS Level hängt damit zusammen, in welche Richtung der Datenfluss geht. Wenn der Client eine Nachricht an den Server publiziert, ist das QoS Level von dem Level der jeweiligen Nachricht abhängig, das der Client entsprechend für diese eingestellt hat. In dem Fall, dass der Server eine Nachricht an den Client schickt, nutzt er das QoS Level, dass der Client zuvor bei der Subskription festgelegt hat. Dies bedeutet, dass das Level bei diesem Faktum herabgestuft wird, wenn der Client mit einem niedrigeren Level subskribiert hat.

*QoS 0 – Höchstens einmal*
Das niedrigste QoS Level ist null. Hier ist die Garantie, dass die Nachricht verlässlich ankommt, auf die Funktionalitäten des zugrundeliegenden TCP-Protokolls beschränkt, denn die PUBLISH Message wird von keinem zusätzlichen Paket bestätigt (siehe Abbildung 30). Die Nachricht kommt beim Empfänger einmal oder gar nicht an. Um das QoS Level 0 zu erreichen, muss der Sender ein PUBLISH Control Packet mit den Flags QoS=0 und DUP=0 senden. Bei dieser Option wird auch von *fire and forget* gesprochen, der Bedeutung nach also abschicken und fertig. Empfehlenswert ist die niedrigste Stufe dann, wenn ein stabiles Netz vorhanden ist oder auf das Risiko eingegangen werden kann, eine Nachricht zu verlieren, beispielsweise bei der regelmäßigen Übertragung von Temperaturwerten.


*Abbildung 30: Quality of Service Level 0 [61]*

*QoS 1 – Mindestens einmal*

Dieses Level sichert der Übertragung zu, dass eine Nachricht beim Empfänger mindestens einmal ankommt. Dafür schickt der Sender ein PUBLISH Packet mit einem einzigartigen Packet Identifier mit den Flags QoS=1 und DUP=0, wenn eine neue Application Message zu versenden ist. In diesem Zustand wird die ursprüngliche PUBLISH Message als *nicht bestätigt* zwischengespeichert, bis der Sender die Bestätigung (PUBACK Control Packet) vom Empfänger empfängt. Im Falle einer Nicht-Bestätigung durch den Empfänger wird die Nachricht nach einem Timeout erneut gesendet. Die Nachricht kann aus diesem Grund auch mehrmals beim Empfänger ankommen. Ein Fall, wo das QoS Level Anwendung finden kann, ist, wenn eine Nachricht zwingend beim Empfänger ankommen muss, Duplikate einer Nachricht aber keinen Einfluss nehmen. Im Gegensatz zu QoS 3 bleibt hier die Netzbelastung durch Bestätigungspakete gering.

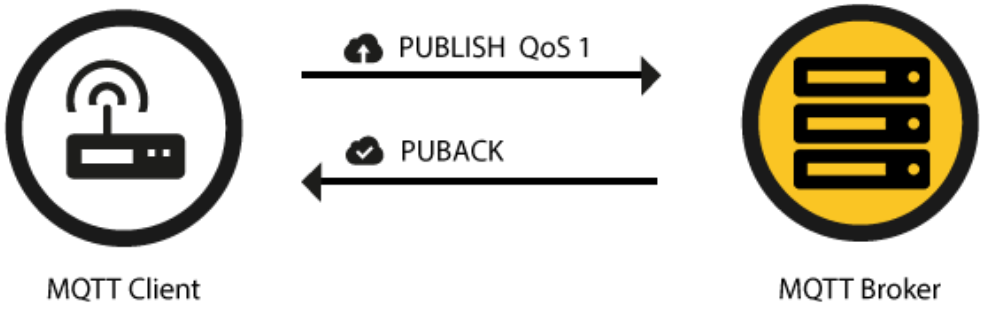



*Abbildung 31: Quality of Service Level 1 [61]*

**PUBACK - Publish Acknowledgement (QoS Level 1)**

*PUBACK* steht für das Publish Acknowledgement Control Packet. Dieses Packet ist die Antwort, beziehungsweise die Bestätigung einer PUBLISH Nachricht mit dem QoS Level 1.

Fixed Header: Die Remaining Length hat im PUBACK Packet den Wert 2 und gibt die Länge des Variable Header an.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *MQTT Control Packet Type (4)* | | | | *Reserved* | | | |
| | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Byte 2... | *Remaining Length (2)* | | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

*Abbildung 32: PUBACK Fixed Header*

Variable Header:

Der Variable Header beinhaltet den Packet Identifier der PUBLISH Nachricht, die bestätigt werden soll.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *Packet Identifier MSB* | | | | | | | |
| Byte 2... | *Packet Identifier LSB* | | | | | | | |

*Abbildung 33: PUBACK Variable Header*

Payload:
Die PUBACK Nachricht hat keinen Payload (siehe Abbildung 14, Seite 26).

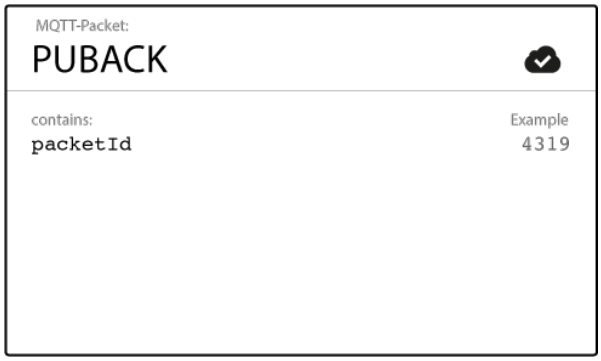



*Abbildung 34: Beispiel einer PUBACK Message [61]*

*QoS 2 – Genau einmal*
Bei dieser Variante wird das höchste Level von QoS bei MQTT angesprochen. Angewendet wird QoS 2 dann, wenn auf keine Nachricht verzichtet werden kann und auch jeweils nur genau eine Nachricht ankommen darf. Der Ablauf des Handshakes findet wie folgt statt: Wenn beim Empfänger ein PUBLISH Control Packet mit einer QoS 2 Nachricht ankommt, bestätigt er diese durch die PUBREC Nachricht. Der Sender, welcher nun die Bestätigung hat, dass die Nachricht angekommen ist, deaktiviert die Wiederholung und bestätigt wiederum dem Empfänger mit der PUBREL Message, dass der Übertragungsvorgang abgeschlossen ist. Die zwischengespeicherte Transaktion wird vom Empfänger beendet und mit der PUBCOMP Nachricht an den Sender quittiert (siehe Abbildung 35). Aufgrund des Overheads durch die drei Packets PUBREC, PUBREL und PUBCOMP, die für die Bestätigung beim Sender sorgen, ist mit einer leicht erhöhten Netzbelastung zu rechnen.

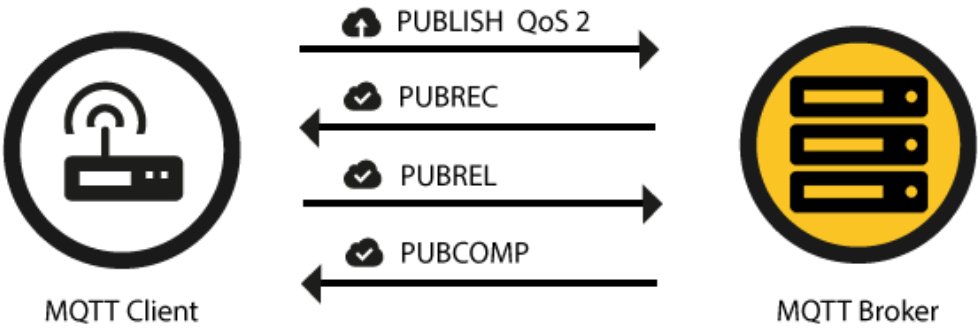



*Abbildung 35: Quality of Service Level 2 [61]*

**PUBREC – Publish Received (QoS Level 2, Teil 1)**
Das *PUBREC Control Packet* ist das *zweite* Nachrichtenpaket nach der PUBLISH Nachricht mit dem QoS Level 2. Sie bestätigt, dass die PUBLISH Nachricht angekommen ist.

Fixed Header: Die Remaining Length hat im PUBREC Packet den Wert 2 und gibt die Länge des Variable Header an.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *MQTT Control Packet Type (5)* | | | | *Reserved* | | | |
| | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| Byte 2... | *Remaining Length (2)* | | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

*Abbildung 36: PUBREC Fixed Header*

Variable Header:
Der Variable Header beinhaltet den Packet Identifier der PUBLISH Nachricht, die bestätigt werden soll.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *Packet Identifier MSB* | | | | | | | |
| Byte 2... | *Packet Identifier LSB* | | | | | | | |

*Abbildung 37: PUBREC Variable Header*

Payload:
Die PUBREC Nachricht hat keinen Payload (siehe Abbildung 14, Seite 26).

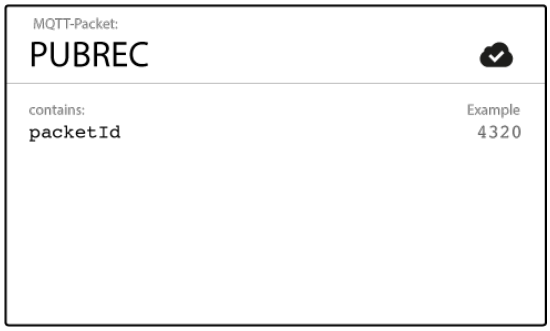



*Abbildung 38: Beispiel einer PUBREC Message [61]*

**PUBREL – Publish Release (QoS Level 2, Teil 2)**
Das *PUBREL Control Packet* ist das *dritte* Control Packet einer PUBLISH Nachricht mit dem QoS Level 2. Sie bestätigt, dass die PUBREC Nachricht angekommen ist (siehe Abbildung 35, Seite 36).

Fixed Header: Die Remaining Length hat im PUBREL Packet den Wert 2 und gibt die Länge des Variable Header an. Im ersten Byte an der Stelle von Bit zwei muss der Wert *1* betragen, damit der Server die Verbindung nicht schließt.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *MQTT Control Packet Type (6)* | | | | *Reserved* | | | |
| | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| Byte 2... | *Remaining Length (2)* | | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

*Abbildung 39: PUBREL Fixed Header*

Variable Header:
Der Variable Header beinhaltet den Packet Identifier der PUBREC Nachricht, die bestätigt werden soll.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *Packet Identifier MSB* | | | | | | | |
| Byte 2... | *Packet Identifier LSB* | | | | | | | |

*Abbildung 40: PUBREL Variable Header*

Payload:
Die PUBREL Nachricht hat keinen Payload (siehe Abbildung 14, Seite 26).

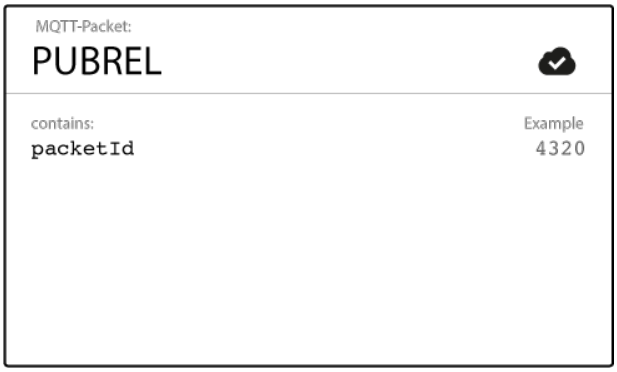



*Abbildung 41: Beispiel einer PUBREL Message [61]*

### PUBCOMP – Publish Complete (QoS Level 2, Teil 3)

Das *PUBCOMP Control Packet* ist das *vierte* und *letzte* Antwortpaket einer PUBLISH Nachricht mit dem QoS Level 2. Sie bestätigt, dass die PUBREL Nachricht angekommen ist (siehe Abbildung 35, Seite 36) und sagt aus, dass das Publishing komplettiert wurde.

Fixed Header: Die Remaining Length hat im PUBCOMP Packet den Wert 2. Die an Position Byte eins Bit null bis drei sind reserviert und müssen 0 sein.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *MQTT Control Packet Type (7)* | | | | *Reserved* | | | |
| | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| Byte 2... | *Remaining Length (2)* | | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

*Abbildung 42: PUBCOMP Fixed Header*

Variable Header:
Der Variable Header beinhaltet den Packet Identifier der PUBREL Nachricht, die bestätigt werden soll.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *Packet Identifier MSB* | | | | | | | |
| Byte 2... | *Packet Identifier LSB* | | | | | | | |

*Abbildung 43: PUBCOMP Variable Header*

Payload:
Die PUBCOMP Nachricht hat keinen Payload (siehe Abbildung 14, Seite 26).

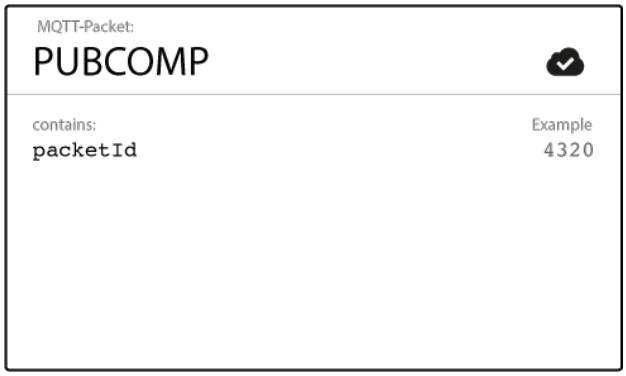



*Abbildung 44: Beispiel einer PUBCOMP Message [61]*

### 3.3.9. Subskriptionen zu einem Topic

**SUBSCRIBE**
Das *SUBSCRIBE Control Packet* wird von einem Client an den Server geschickt, um eine oder mehrere Subskriptionen unter bestimmten Topic Names durchzuführen. Der Server sendet PUBLISH Nachrichten daraufhin an den empfangenden Client, um Application Messages weiterzuleiten, die von anderen Clients unter diesem Topic publiziert worden sind. In einer SUBSCRIBE Nachricht wird zusätzlich für jede Subskription das maximale QoS Level festgelegt, mit dem der Server die Nachrichten an den Client schicken darf. Als Bestätigung erwartet der Client ein SUBACK Packet vom Server (siehe Abbildung 45). Weitere Einzelheiten zu Subskription, Topics und Filtern werden zuvor in Kapitel 3.3.2 und 3.3.3 behandelt.

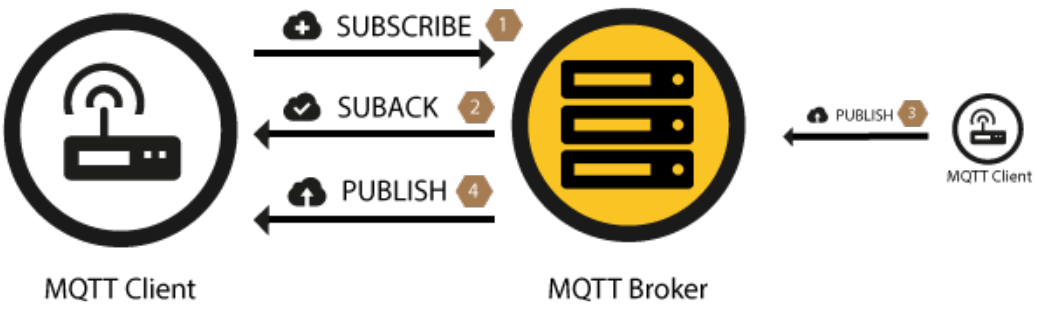



*Abbildung 45: Subskription über SUBSCRIBE und SUBACK [62]*

Fixed Header:
Im ersten Byte an der Stelle von Bit zwei muss der Wert *1* betragen, damit der Server die Verbindung nicht schließt. Die Remaining Length beträgt an dieser Stelle die Länge des Variable Header plus die Länge des Payloads.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *MQTT Control Packet Type (8)* | | | | *Reserved* | | | |
| | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Byte 2... | *Remaining Length* | | | | | | | |

*Abbildung 46: SUBSCRIBE Fixed Header*

Variable Header:
Der Variable Header beinhaltet einen Packet Identifier, der auf den Wert 10 gesetzt ist.

| Bit | Beschreibung | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| *Packet Identifier* | | | | | | | | | |
| Byte 10 | Packet Identifier MSB (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Byte 11 | Packet Identifier LSB (10) | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

*Abbildung 47: SUBSCRIBE Variable Header*

Payload:
Der Payload eines SUBSCRIBE Packets beinhaltet eine Liste mit Topic Filtern, zu denen der Client subskribieren möchte. Die Topic Filter werden als UTF-8 Strings übertragen. Jeder Topic Filter in der Liste bekommt ein Byte *Requested QoS* zugeordnet. Die Bits zwei bis sechs des Requested QoS sind für zukünftige Versionen reserviert.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| *Topic Filter* | | | | | | | | |
| Byte 1 | *Length MSB* | | | | | | | |
| Byte 2 | *Length LSB* | | | | | | | |
| Byte 3..N | *Topic Filter* | | | | | | | |
| *Requested QoS* | | | | | | | | |
| | Reserved | | | | | | QoS | |
| Byte N+1 | 0 | 0 | 0 | 0 | 0 | 0 | X | X |

*Abbildung 48: SUBSCRIBE Payload*

Im folgenden Beispiel eines SUBSCRIBE Payloads sind die Topics `topic/1` mit QoS Level 1 und `topic/2` mit QoS Level 2 als Topic Names vorhanden.

| Bit | Beschreibung | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| *Topic Filter* | | | | | | | | | |
| Byte 1 | Länge MSB (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Byte 2 | Länge LSB (7) | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| Byte 3 | 't' (0x74) | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| Byte 4 | 'o' (0x6f) | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| Byte 5 | 'p' (0x70) | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| Byte 6 | 'i' (0x69) | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| Byte 7 | 'c' (0x63) | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| Byte 8 | '/' (0x2f) | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| Byte 9 | '1' (0x31) | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| *Requested QoS* | | | | | | | | | |
| Byte 10 | Requested QoS (1) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| *Topic Filter* | | | | | | | | | |
| Byte 11 | Länge MSB (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Byte 12 | Länge LSB (7) | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| Byte 13 | 't' (0x74) | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| Byte 14 | 'o' (0x6f) | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| Byte 15 | 'p' (0x70) | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| Byte 16 | 'i' (0x69) | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| Byte 17 | 'c' (0x63) | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| Byte 18 | '/' (0x2f) | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| Byte 19 | '2' (0x32) | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |
| *Requested QoS* | | | | | | | | | |
| Byte 20 | Requested QoS (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

*Abbildung 49: SUBSCRIBE Payload Beispiel*

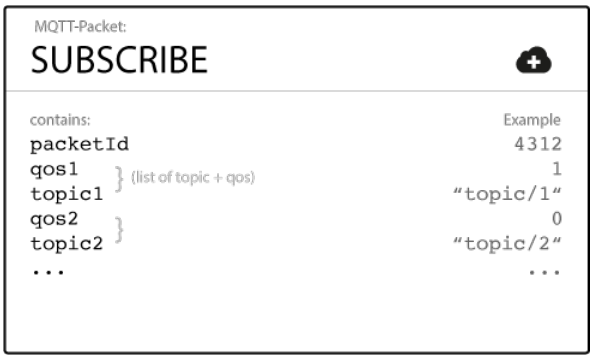



*Abbildung 50: Beispiel einer SUBSCRIBE Message [62]*

SUBACK

Das *SUBACK Control Packet* ist die Antwort-Nachricht vom Server an den Client auf die SUBSCRIBE Message und quittiert diese. Diese Nachrichtenart beinhaltet den gleichen Packet Identifier wie die ursprüngliche Nachricht und eine Liste von Return Codes, welche die Maxima des jeweiligen QoS Levels einer Subskription beinhalten.

Fixed Header:

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *MQTT Control Packet Type (9)* | | | | *Reserved* | | | |
| | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| Byte 2... | *Remaining Length* | | | | | | | |

*Abbildung 51: SUBACK Fixed Header*

Variable Header:

Der Variable Header beinhaltet den Packet Identifier der SUBSCRIBE Nachricht, die bestätigt werden soll.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *Packet Identifier MSB* | | | | | | | |
| Byte 2... | *Packet Identifier LSB* | | | | | | | |

*Abbildung 52: SUBACK Variable Header*

Payload:

Der Return Code im Payload (siehe Abbildung 55) bezieht sich auf den Topic Filter im SUBSCRIBE Packet, das bestätigt werden soll. Wenn in einer SUBSCRIBE Nachricht mehrere Topics angegeben wurden, muss die Liste an Return Codes der Reihenfolge der ursprünglichen Nachricht entsprechen.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| | Return Code | | | | | | | |
| Byte 1 | X | 0 | 0 | 0 | 0 | 0 | X | X |

*Abbildung 53: SUBACK Payload*

In folgendem Beispiel wird eine Auflistung von Return Codes von drei Topics einer SUBSCRIBE Anfrage gezeigt.

| Bit | Beschreibung | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| Byte 1 | Success – Maximum QoS 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Byte 2 | Success – Maximum QoS 2 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| Byte 3 | Failure | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

*Abbildung 54: SUBACK Payload Beispiel*

| Wert | Return Code Antwort |
|---|---|
| 0 | 0x00 Success – Maximum QoS 0 |
| 1 | 0x01 Success – Maximum QoS 1 |

| | |
|---|---|
| 2 | 0x02 Success – Maximum QoS 2 |
| 128 | 0x80 Failure |

*Abbildung 55: SUBACK Return Codes*

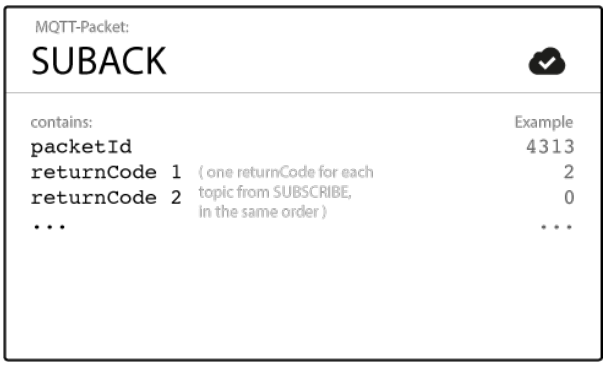



*Abbildung 56: Beispiel einer SUBACK Message [62]*

### UNSUBSCRIBE

Um eine Subskription rückgängig zu machen, also ein Abonnement zu einem Topic zu beenden, sendet der Client ein *UNSUBSCRIBE Control Packet* zum Server.

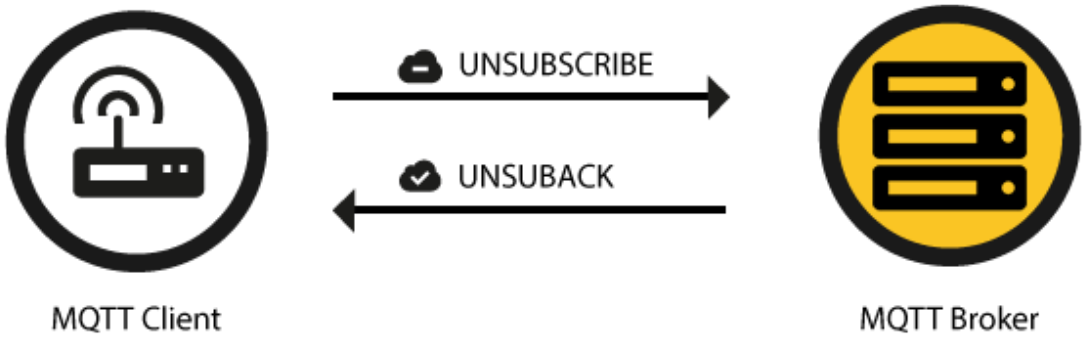



*Abbildung 57: Un-Subskribierung UNSUBSCRIBE und UNSUBACK [62]*

Fixed Header:

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *MQTT Control Packet Type (10)* | | | | *Reserved* | | | |
| | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| Byte 2... | *Remaining Length* | | | | | | | |

*Abbildung 58: UNSUBSCRIBE Fixed Header*

Variable Header:

Der Variable Header beinhaltet einen Packet Identifier, um eine Zuordnung der Nachricht exakt vornehmen zu können.

| Bit | Beschreibung | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| *Packet Identifier* | | | | | | | | | |
| Byte 10 | Packet Identifier MSB | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Byte 11 | Packet Identifier LSB | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |

*Abbildung 59: UNSUBSCRIBE Variable Header*

Payload:
Der Payload eines UNSUBSCRIBE Packets beinhaltet eine Liste mit Topic Filtern, zu denen der Client die Subskription beenden möchte. Die Vorgehensweise ist hier analog zur SUBSCRIBE Message.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| *Topic Filter* | | | | | | | | |
| Byte 1 | *Length MSB* | | | | | | | |
| Byte 2 | *Length LSB* | | | | | | | |
| Byte 3..N | *Topic Filter* | | | | | | | |

*Abbildung 60: UNSUBSCRIBE Payload*

Das Beispiel zeigt den Payload eines UNSUBSCRIBE Packets eines Clients, der die Subskription zu den Topics `topic/1` und `topic/2` beenden möchte.

| Bit | Beschreibung | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|
| *Topic Filter* | | | | | | | | | |
| Byte 1 | Länge MSB (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Byte 2 | Länge LSB (7) | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| Byte 3 | ‘t’ (0x74) | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| Byte 4 | ‘o’ (0x6f) | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| Byte 5 | ‘p’ (0x70) | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| Byte 6 | ‘i’ (0x69) | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| Byte 7 | ‘c’ (0x63) | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| Byte 8 | ‘/’ (0x2f) | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| Byte 9 | ‘1’ (0x31) | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 |
| *Topic Filter* | | | | | | | | | |
| Byte 10 | Länge MSB (0) | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Byte 11 | Länge LSB (7) | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| Byte 12 | ‘t’ (0x74) | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 |
| Byte 13 | ‘o’ (0x6f) | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 |
| Byte 14 | ‘p’ (0x70) | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| Byte 15 | ‘i’ (0x69) | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| Byte 16 | ‘c’ (0x63) | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| Byte 17 | ‘/’ (0x2f) | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 |
| Byte 18 | ‘2’ (0x32) | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |

*Abbildung 61: UNSUBSCRIBE Payload Beispiel*


*Abbildung 62: Beispiel einer UNSUBSCRIBE Message [62]*

### UNSUBACK

*UNSUBACK* ist die Bestätigung der UNSUBSCRIBE Nachricht. Diese wird vom Server an den Client geschickt. Die Topic Filter, die in der UNSUBSCRIBE Nachricht angegeben werden, werden Zeichen für Zeichen mit den aktuell gespeicherten Topics verglichen. Wenn ein übereinstimmender Topic Name gefunden wird, wird die Subskription beendet. Auch wenn kein passendes Topic ausfindig gemacht werden kann und keine Subskription gelöscht wird, muss ein UNSUBACK Control Packet gesendet werden.

Fixed Header:

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *MQTT Control Packet Type (11)* | | | | *Reserved* | | | |
| | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| Byte 2... | *Remaining Length (2)* | | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

*Abbildung 63: UNSUBACK Fixed Header*

Variable Header:
Wie gehabt beinhaltet das Control Packet dieses Bestätigungs-Paket den Packet Identifier der ursprünglichen Nachricht, die bestätigt werden soll.

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *Packet Identifier MSB* | | | | | | | |
| Byte 2... | *Packet Identifier LSB* | | | | | | | |

*Abbildung 64: UNSUBACK Variable Header*

Payload:
Das UNSUBACK Packet hat keinen Payload (siehe Abbildung 14, Seite 26).

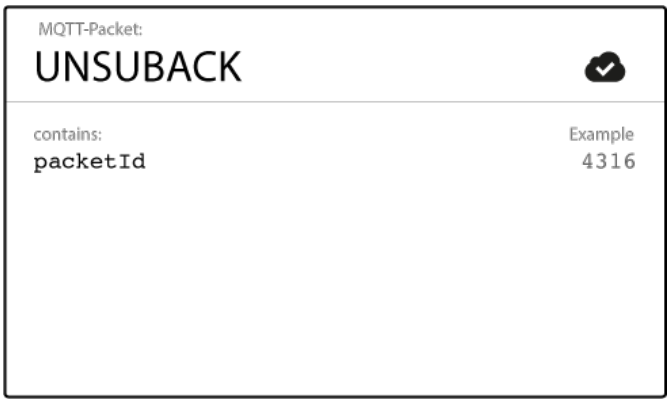



*Abbildung 65: Beispiel einer UNSUBACK Message [62]*

### 3.3.10. Ping

#### PINGREQ

Ein *PINGREQ Control Packet* wird vom Client zum Server gesendet und wird dazu benutzt, dem Server zu zeigen, dass der Client noch *alive* ist, wenn gerade keine Control Packets vom Client zum Server geschickt werden. Außerdem kann durch ein Request festgestellt werden, ob der Server noch erreichbar ist. Auf ein PINGREQ muss immer eine Antwort des Servers erfolgen, indem er mit einem PINGRESP Control Packet antwortet.

Fixed Header:

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *MQTT Control Packet Type (12)* | | | | *Reserved* | | | |
| | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Byte 2... | *Remaining Length (0)* | | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

*Abbildung 66: PINGREQ Fixed Header*

Variable Header:
Das PINGREQ Packet hat keinen Variable Header.

Payload:
Das PINGREQ Packet hat keinen Payload (siehe Abbildung 14, Seite 26).

#### PINGRESP

Das *PINGRESP Control Packet* ist die Antwort auf die PINGREQ Anfrage, die der Server an den Client schickt und indiziert, dass der Server erreichbar ist.

Fixed Header:

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *MQTT Control Packet Type (13)* | | | | *Reserved* | | | |
| | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| Byte 2... | *Remaining Length (0)* | | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

*Abbildung 67: PINGRESP Fixed Header*

Variable Header:
Das UNSUBACK Packet hat keinen Variable Header.

Payload:
Das UNSUBACK Packet hat keinen Payload (siehe Abbildung 14, Seite 26).

### 3.3.11. Verbindungsabbau

#### DISCONNECT

Um eine bestehende Verbindung zu trennen, wird das *DISCONNECT Control Packet* vom Client zum Server geschickt. Diese Benachrichtigung teilt dem Server mit, dass der Client die Verbindung abbaut. Nach dem Senden dieses Packets darf der Client keine weiteren Control Packets senden. Beim Empfangen des Packets sollte der Server die Verbindung schließen, insofern der Client das noch nicht gemacht hat, und alle *Will Messages* (siehe 3.3.12 *Last Will*) verwerfen, die etwas mit der Verbindung zu diesem Client zu tun hatten.

Fixed Header:

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Byte 1 | *MQTT Control Packet Type (14)* | | | | *Reserved* | | | |
| | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| Byte 2... | *Remaining Length (0)* | | | | | | | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

*Abbildung 68: DISCONNECT Fixed Header*

Variable Header:
Das UNSUBACK Packet hat keinen Variable Header.

Payload:
Das UNSUBACK Packet hat keinen Payload (siehe Abbildung 14, Seite 26).

### 3.3.12. Weitere Features

#### Persistent Session

Bei dem Verlust der Verbindung von einem Client zum Broker gehen beim Broker Informationen über die Subskriptionen des betreffenden Clients verloren. Ein Beispiel wäre hier, welche Topics dieser abonniert hatte. Bei dem Wiederverbinden mit dem gleichen Client müsste sich dieser zu dem Topic erneut subskribieren. Die sogenannte *Persistant Session* ermöglicht es, über den Verbindungszeitraum hinaus Informationen serverseitig zu speichern. Identifiziert wird eine *Session* über den *Client Identifier,* der beim Aufbau der Verbindung durch ein CONNECT Control Packet mit dem *Clean Session Flag* geschickt wird. Gerade bei Geräten mit limitierten Ressourcen macht eine Speicherung der Session Sinn, denn nach einem Verlust der Verbindung müsste dieses erst wieder sämtliche Topics abonnieren.

Nur wenn das Clean Session Flag auf *false* steht, wird beim Verbindungsaufbau eine Persistent Session erstellt. Das Antwort-Paket CONNACK vom Server teilt dem Client durch das *Session Present Flag* mit, ob eine Session gespeichert wurde oder nicht. Vom Server werden dabei folgende Informationen mit der Session abgespeichert:

- Sämtliche Subskriptionen
- Alle Nachrichten mit den QoS Leveln 1-2, die noch nicht vom Client bestätigt wurden

- Jegliche QoS 1 und QoS 2 Nachrichten, die noch nicht an den Client bestätigt wurden
- Alle Nachrichten mit QoS Level 1 und 2, die der Client verpasst hat, während er offline war und weitergeleitet werden müssen

Wenn vom Client eine Persistent Session erwünscht ist, ist dieser auch dazu verpflichtet, einige Informationen zur aktuellen Session zu speichern. Um das Soll von Nachrichten mit dem QoS Level 1 und 2 zu erfüllen, werden Nachrichten beim Client gespeichert, die

- noch nicht vom Server bestätigt wurden
- noch an den Server bestätigt werden müssen [63]

### Retained Messages

Der Hintergrund von *Retained Messages* ist, dass Clients nichts von dem Dasein eines anderen Clients wissen. Ein publizierender Client kann somit auch nicht garantieren, dass eine Nachricht beim End-Knoten ankommt. Das Gleiche gilt auch für die subskribierenden Clients. Sie wissen nicht, ob und wann ein Publisher eine Nachricht gesendet hat. Eine Nachricht, die über PUBLISH an den Broker geschickt wird und mit dem sogenannten *Retain Flag* markiert ist, bewirkt, dass immer die letzte eintreffende Nachricht zu einem Topic beim Broker gespeichert wird.

Jeder Client, der sich zu diesem Topic zu einer mit dem Retain Flag versehenen Nachricht subskribiert, bekommt nun direkt die zuletzt beim Broker eingetroffene Nachricht, auch wenn der Sender zur gleichen Zeit keine Nachrichten verschickt. Retained Messages ermöglichen es einem neu hinzukommenden Client unverzüglich, eine Nachricht beim Broker abzufragen. Vor allem bei Anwendungen, die einen Status abfragen wollen, der vom Publisher aber nur in regelmäßigen zeitlichen Intervallen veröffentlicht wird, ist diese Art von Nachricht von Vorteil.

Um eine Retained Message beim Server zu löschen, muss eine Nachricht mit dem Retained Flag gesetzt an selbigen unter gleichbleibendem Topic gesendet werden. Eine Voraussetzung, dass die Nachricht gelöscht wird, ist, dass der Payload der Nachricht eine Größe von null Byte aufweisen muss [64].

### Last Will and Testament

MQTT wird häufig in limitierten Netzwerken benutzt, bei denen Ausfälle von einzelnen Geräten keine Seltenheit sind. Gründe dafür können Störfaktoren der Funkverbindung oder eine entleerte Batterie eines Devices sein. Um mit diesem Problem umzugehen, hat MQTT den Mechanismus *Last Will and Testament (LWT)* integriert. Diese Funktionalität sendet bei dem Verlust einer Verbindung eines Clients eine Benachrichtigung an andere Clients.

Schon beim Aufbau einer Verbindung zum Broker wird festgelegt, ob eine solche Nachricht bei diesem Use-Case verschickt wird, unter welchem Topic, mit welchem QoS Level und welchem Payload dies geschieht. Der Broker speichert diese Nachricht, und im Falle eines Verbindungsverlusts sendet dieser die Nachricht an alle Clients, die sich zu diesem Topic subskribiert haben. Wenn sich ein Client korrekterweise über eine DISCONNECT Nachricht vom Broker abmeldet, wird die Last Will Nachricht verworfen.

Der letzte Wille wird beim Verbindungsaufbau mit dem CONNECT Packet versendet. Die *Will Flag, Will Retain Flag* und *Will QoS Flag* (siehe Abbildung 19, Seite 29) definieren, wie der Last Will ausgeprägt ist. Im Payload werden der *Will Topic* und die *Will Message* festgeschrieben [65].

### MQTT Security

Die Sicherheitsarchitektur in MQTT beinhaltet eigene Sicherheitsmechanismen, bereits vorhandene Lösungen werden ebenfalls unterstützt. Diese Optionen beziehen sich bei MQTT auf die Authentifizierung und Autorisierung von Benutzern und Geräten, auf die Integrität und den Datenschutz bei den Control Packets und den Application Data [54].

Auf der Netzwerkschicht wird bei der Verwendung einer WLAN-Verbindung bereits eine Verschlüsselung über das WLAN-Netz hinweg vorgenommen. Ein Virtuelles Privates Netzwerk bietet des Weiteren die Sicherheit, dass Daten nur von autorisierten Clients empfangen werden können, indem die Kommunikation zwischen Knoten und Broker über einen sicheren Tunnel geschieht. Auf der Transport-Ebene sorgen *TLS (Transport Security Layer)* und *SSL (Secure Socket Layer)* dafür, dass die Verbindung abgesichert wird. Auf dem Application Layer findet der von MQTT bereitgestellte Sicherheitsmechanismus durch Authentifizierung und Autorisierung Anwendung [40].

Für die Authentifizierung und Autorisierung von Clients beim Server sind beim CONNECT Packet Felder für Benutzername und Passwort im Payload vorgesehen. Die Flags dafür werden im Variable Header dieses Pakets gesetzt (siehe Abbildung 19, Seite 29). Ohne diese Authentifizierung könnte jeder beliebiger Client Nachrichten an den Broker publizieren. Der Grund dafür liegt darin, dass in der Standard-Konfiguration des Brokers der Parameter *allow anonymous* auf *true* gesetzt ist. Für die Verwendung dieses Sicherheitsmechanismus muss eine Passwort-Datei erstellt werden und der zuvor beschriebene Parameter auf *false* gesetzt werden [40].

Für eine Anwendung, die einer zusätzlicher Absicherung bedarf, gibt es die Möglichkeit, eine Authentisierung über speziell festgelegte Client-IDs vorzunehmen. Hierfür wird jede gesicherte Client-ID mit einem Präfix versehen, der zuvor in der Konfigurationsdatei festgelegt wird. Anschließend werden nur noch Clients vom Broker akzeptiert, bei denen das definierte Präfix zu Beginn der Client-ID steht.

Für ein stärker abgesichertes MQTT-Netzwerk steht eine weitergehende Verschlüsselungsmethode zur Verfügung. Damit während der Übertragung keine Daten abgegriffen werden können, wird der Verschlüsselungsstandard TLS (SSL) verwendet, bei dem ein sicherer Kanal zwischen Knoten und Server aufgebaut wird. Möglich ist hier die Benutzung des Public-Key-Verfahrens oder eines symmetrischen Verschlüsselungsverfahrens, wie das Pre-Shared-Key-Verfahren *(PSK)*. Bei dem PSK-Verfahren wird der Schlüssel vorher vereinbart. Diese Variante hat den Vorteil, keinen Schlüsselserver zwischenschalten zu müssen. Vor dem Aufbau der Verbindung muss allerdings der Schlüssel ausgetauscht werden. Für die meisten IoT-Anwendungen ist das PSK-Verfahren geeignet, da ein Austausch des Schlüssels im Vorhinein möglich ist. Es ist im Voraus zu prüfen, ob die betreffenden Clients eine Unterstützung von TLS anbieten. Weiterhin ist zu beachten, dass durch die Verwendung von MQTT über TLS Overhead in Form von höherer Bandbreiten-Nutzung (TLS-Handshake) und eine höhere Prozessorauslastung entsteht.

## 3.4. Verbreitung

MQTT hat sich bei der Recherche zu dieser Arbeit als mittlerweile verbreitetes Nachrichtenprotokoll präsentiert. Es wird von großen Konzernen wie Facebook und im Bereich IoT auch von Amazon und IBM verwendet [9] [66] [46]. Ein weiterer Indikator dafür ist die hohe Trefferquote einer Google-Suche mit 4.780.000 Treffern zu dem Suchbegriff „MQTT". Bei der Auswertung des Begriffs bei *Google Trends* (siehe Abbildung 69) wird ersichtlich, dass das Thema ein stetig wachsendes Interesse aufweist. Nach einer Suche auf dem Entwickler-Forum *stackoverflow.com* werden zu diesem Stichwort etwa 6400 Treffer angezeigt, was sich während der bisherigen Arbeit bei Fragen zu diesem Protokoll als durchaus nützlich herausstellte.


*Abbildung 69: Google Trends zu dem Suchbegriff „MQTT" [67]*

## 3.5. Unterstützte Systeme/Frameworks

### 3.5.1. Systeme (IoT-Plattformen/MaaS- und PaaS-Dienste)

Cloud-Services nehmen mittlerweile in vielen Bereichen einen hohen Stellenwert ein. Viele kommerzielle Anbieter bringen inzwischen Lösungen auf den Markt, wo ganze Anwendungen in der Cloud entwickelt und deployed werden. So ist es auch im Bereich IoT zu beobachten. Gerade, wenn Endgeräte nicht mehr nur im lokalen Netz, sondern über das Internet miteinander kommunizieren sollen, ist ein sich im Internet befindlicher Server unabdingbar. Mit sogenannten PaaS-Diensten (Platform as a Service) lassen sich die dort anfallenden Daten verwalten und weiterverarbeiten. Die folgend aufgelisteten IoT-Plattformen sind Lösungen, die Daten in einem IoT-Netzwerk auch über das Internet über das MQTT Protokoll weiterleiten, verarbeiten und darstellen können:

| Name | Beschreibung | Informationen abrufbar unter | Lizenz |
|---|---|---|---|
| *Eurotech Everywhere Cloud* | M2M IoT Plattform mit Broker | *eurotech.com/en/products/software+services/everyware+cloud+m2m+platform* | kommerziell |
| *Xively* | Cloud-Service für IoT-Daten mit MQTT-Support | *xively.com/xively-iot-platform* | kommerziell |
| *Litmus Automation Loop* | IoT Cloud Plattform mit Support für MQTT | *litmusautomation.com/loop-cloud/* | kommerziell |

| | | | |
|---|---|---|---|
| *Solace Cloud beta* | IoT Plattform mit multi-Protokoll Support | *cloud.solace.com* | kommerziell, freier Zugang |
| *Thingscale IoT message broker* | IoT Plattform mit multi-Protokoll Support | *thingscale.io* | kommerziell, kostenloses Tryout für 30 Tage |
| *flespi* | cloudbasierter Online-Broker | *flespi.com/mqtt-broker* | kommerziell, auch kostenlose Version |
| *WSO2 IoT Server* | IoT Plattform mit MQTT Support | *wso2.com/iot/features* | Open Source |
| *CloudMQTT* | IoT Message Broker basierend auf Mosquitto | *cloudmqtt.com* | kommerziell, frei mit Einschränkungen |
| *Carriots* | PaaS-Dienst für IoT mit MQTT Support | *carriots.com* | kommerziell, frei mit Einschränkungen |
| *Beebotte* | Cloud Platform as a Service mit MQTT Support | *beebotte.com/home* | kommerziell, frei mit Einschränkungen |
| *IBM Watson IoT* | IoT Cloud Plattform mit MQTT Support | *ibm.com/internet-of-things* | kommerziell |
| *Adafruit IO* | PaaS-Dienst für IoT mit MQTT Support | *io.adafruit.com* | kommerziell, frei mit Einschränkungen |
| *AWS IoT* | Cloud Plattform zum Sammeln, Verarbeiten und Analysieren von Daten | *aws.amazon.com/de/iot/* | kommerziell |
| *Bosch IoT Suite* | Cloud Services für IoT mit Unterstützung von MQTT | *bosch-iot-suite.com* | Open Source |
| *SAP Cloud Plattform IoT* | *Cloud Service für IoT mit Support für MQTT* | *help.sap.com/viewer/product/SAP_CP_IOT_CF/Cloud/en-US* | kommerziell |
| *Thingspeak* | IoT Plattform zum Analysieren und Visualisieren von Daten | *thingspeak.com* | kommerziell |

*Abbildung 70: IoT Cloud Lösungen für MQTT*

### 3.5.2. Broker

Broker sind bei der Kommunikation über MQTT das Herzstück des Systems. Sie wickeln sämtlichen Ablauf des Informationsflusses ab, indem sie sich um das Weiterleiten der Daten und das Administrieren der Clients kümmern. Deswegen ist es wichtig, vor der Wahl eines Brokers dessen Anforderungen und Vor- und Nachteile zu kennen. Anschließend aufgelistet sind Broker, die nicht cloudbasiert, lokal installiert verwendet werden können:

| Name | Beschreibung | Support für | Informationen abrufbar unter | Lizenz |
|---|---|---|---|---|
| *MQTTnet* | .NET-Bibliothek für MQTT-Kommunikation, enthält Client und Broker | .NET, UWP, Mono, Xamarin | github.com/chkr1011/MQTTnet/ | Open Source MIT License |
| *Bevywise MQTTRoute* | Broker für hoch performante Kommunikation mit vielen Clients | Windows, Linux, Mac | bevywise.com/mqtt-broker/ | kommerziell |
| *Moquette* | leichtgewichtiger MQTT Broker | Java-Plattform | *github.com/andsel/moquette* | Open Source |
| *Mosquitto* | Implementierung eines MQTT Brokers und Clients | Windows, Mac, Linux, Raspberry Pi, iOS | *mosquitto.org* | Open Source |
| *Emitter* | frei verfügbarer Broker Service | JavaScript, Python, C#, Go, Java | *emitter.io* | Open Source |
| *EMQ Erlang MQTT Broker* | frei verfügbarer Broker | Linux, Unix, Mac, Windows, Raspberry Pi | *github.com/emqtt/emqttd* | Open Source Lizenz Apache Version 2.0 |
| *RabbitMQ* | Message Broker für mehrere Nachrichtenprotokolle, auch als Cloud-Broker geeignet, RabbitMQ MQTT Adapter Plugin benötigt | Linux, Unix, Windows, Mac, u.v.m | *rabbitmq.com* | Open Source |

| | | | | |
|---|---|---|---|---|
| *Apache ActiveMQ* | Messaging Server für MQTT und AMQP | Java, C, C++, C#, Ruby, Python, PHP, Windows, Unix und Linux | *activemq.apache.org* | Open Source |
| *Apache ActivMQ Artemis* | Messaging System für verschiedene Nachrichtenprotokolle | Java-Plattform | *activemq.apache.org/artemis/index.html* | Open Source |
| *Apache ActivMQ Apollo* | Multi-Protokoll Messaging Broker basierend auf ActiveMQ | Windows, Unix, Linux, Mac | *activemq.apache.org/apollo/index.html* | Open Source |
| *HiveMQ* | MQTT Broker | Linux, Unix, Mac, Windows | *hivemq.com* | kommerziell |
| *Mosca* | kostenloser MQTT Broker erhältlich als Standalone oder node.js Modul | Linux, Unix | *mosca.io* | Open Source MIT License |
| *JoramMQ* | nachrichtenorientierte Middleware von ScaleAgent, stellt u.a. einen MQTT Server zur Verfügung | Java-Plattform | *scalagent.com/en/jorammq-33/technology-36/mqtt-protocol* | kommerizell |
| *Solace Virtual Message Router* | Multi-Protokoll Message Broker mit Cloud Anbindung | Sämtliche Betriebssysteme | *dev.solace.com* | kostenlose Community- und kostenpflichtige Enterprise Edition |
| *VerneMQ* | performanter Broker | Mac, Raspberry Pi, Linux | *vernemq.com* | Open Source Lizenz Apache Version 2.0 |
| *HBMQTT* | Implementierung von Broker und Client | Python, Linux, Unix | *github.com/beer-factory/hbmqtt* | Open Source |
| *Vert.x MQTT Broker* | Einfacher Broker | Java-Plattform | *github.com/GruppoFilippetti/vertx-mqtt-broker* | Open Source |

*Abbildung 71: MQTT Broker für MQTT*

### 3.5.3. Tools und Programme für Clients

Praktische Hilfsmittel beim Arbeiten mit MQTT Kommunikation sind Anwendungen, die eine Implementierung eines MQTT Clients integriert haben. Mit diesen können in der Regel Verbindungsprofile zu Brokern eingestellt, sowie Subskriptionen und das Veröffentlichen von Daten vorgenommen werden. Dies dient dazu, Verbindungen auf einem unkomplizierten Weg zu testen und zu debuggen. Im Folgenden werden einige für unterschiedliche Plattformen (Desktop-Anwendungen, Web-Applikationen, mobile Apps) verfügbare Tools vorgestellt, die dies ermöglichen. Es wird bei einigen Anwendungen überprüft, ob die grundlegenden Anforderungen an die vorgesehene Anwendung gegeben sind. Es wird getestet, ob sie sich mit einem MQTT-Broker verbinden (CONNECT), an ein Topic eine Nachricht publizieren (PUBLISH) und sich zu einem Topic subskribieren können (SUBSCRIBE).

**Desktop:**

- *Paho MQTT Utility* (Grafisches MQTT Client Tool, Linux, Windows, Mac OSX): kostenlos verfügbar unter *eclipse.org/paho/components/tool* / → CONNECT: ja, SUBSCRIBE: ja, PUBLISH: ja

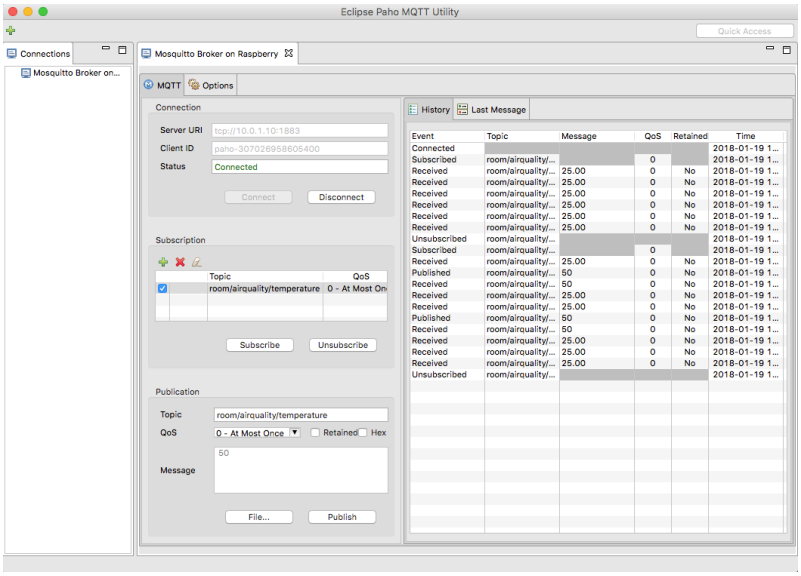



*Abbildung 72: Paho MQTT Utility*

- *MQTT.fx* (siehe 4.3.4 Verwendete Software, Seite 70)

- *mqtt-spy* (Open Source Tool für MQTT Konnektivität basierend auf JavaFX, Linux, Windows, Mac OSX): kostenlos verfügbar unter *github.com/eclipse/paho.mqtt-spy/wiki/Downloads* → CONNECT: ja, SUBSCRIBE: ja, PUBLISH: ja

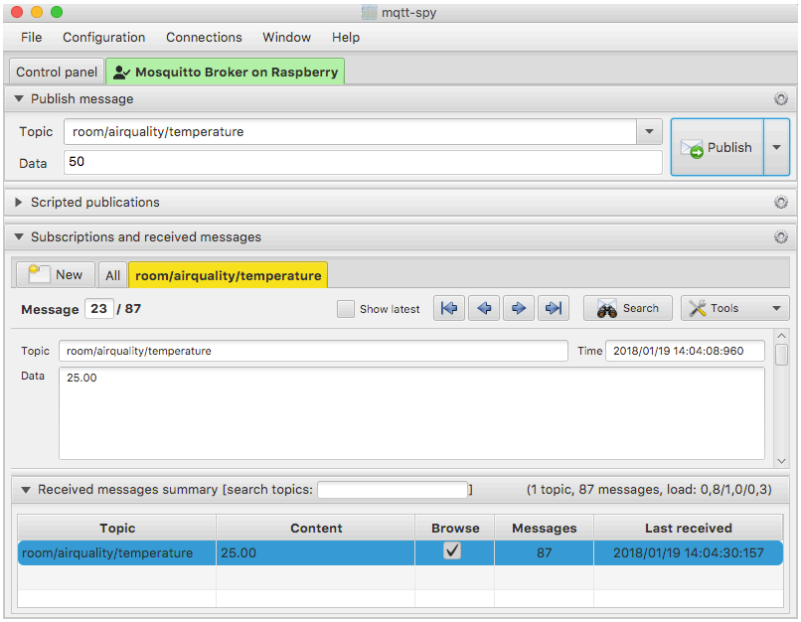



*Abbildung 73: mqtt-spy Java Anwendung*

- *TT3* (MQTT Client Anwendung basierend auf Paho, Windows): Informationen unter *github.com/francoisvdm/TT3* → CONNECT: ja, SUBSCRIBE: ja, PUBLISH: ja

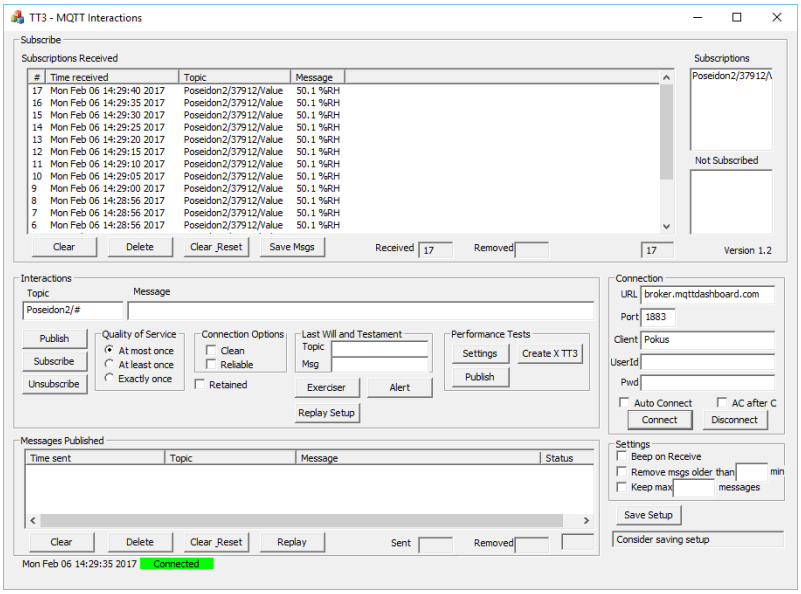



*Abbildung 74: TT3 MQTT Tool*

Die Recherche zu den Desktop-Anwendungen zeigt, dass *sämtliche Anwendungen* zuverlässig funktionieren und eine Möglichkeit bieten, die unterschiedlichen Features von MQTT zu testen.

**Web:**

Anwendungen werden verstärkt plattformunabhängig browserbasiert angeboten. Eine Möglichkeit, Kommunikation über MQTT im Browser zu betreiben, wird durch die Verwendung des Websocket-Protokolls gegeben. Ein Websocket-Protokoll ist ein auf TCP basierendes Kommunikationsprotokoll für Vollduplex-Verbindungen. Die Unterstützung eines MQTT-Websockets wird durch die Verwendung von JavaScript geboten. Für MQTT über Websockets muss der Broker dementsprechend konfiguriert sein. Die Verbindung läuft dann meist über den HTTP-Port 8080 und nicht über den Standard-MQTT-Port 1883. Bei dem Datenaustausch von MQTT Paketen über Websockets werden die betreffenden Pakete in ein Websocket Paket verpackt und zum Client gesendet. Dieser entpackt das Websocket Paket und behandelt dieses als MQTT Paket. Exemplarisch sind an dieser Stelle einige Varianten für die MQTT Kommunikation über Websockets aufgelistet:

- *mqtt-panel* (lokales Web Interface für das Visualisieren von Daten nach einer Subskription): Informationen unter *github.com/fabaff/mqtt-panel*

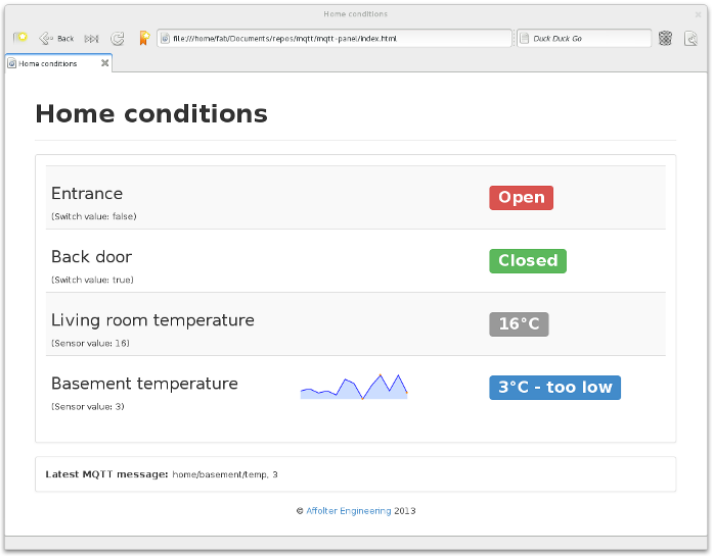



*Abbildung 75: mqtt-panel Web Interface*

- *mqtt-svg-dash* (lokale SVG Seite für die Visualisierung von Daten durch Subskription): Informationen unter *github.com/jpmens/mqtt-svg-dash*

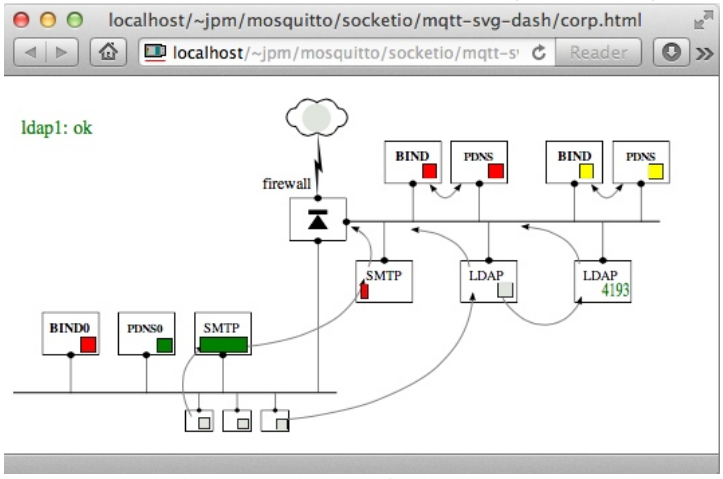



*Abbildung 76: mqtt-svg-dash Oberfläche*

- *HiveMQ Websocket Client* (Websockets Client Showcase): aufrufbar unter *hivemq.com/demos/websocket-client/* → CONNECT: ja, SUBSCRIBE: ja, PUBLISH: ja

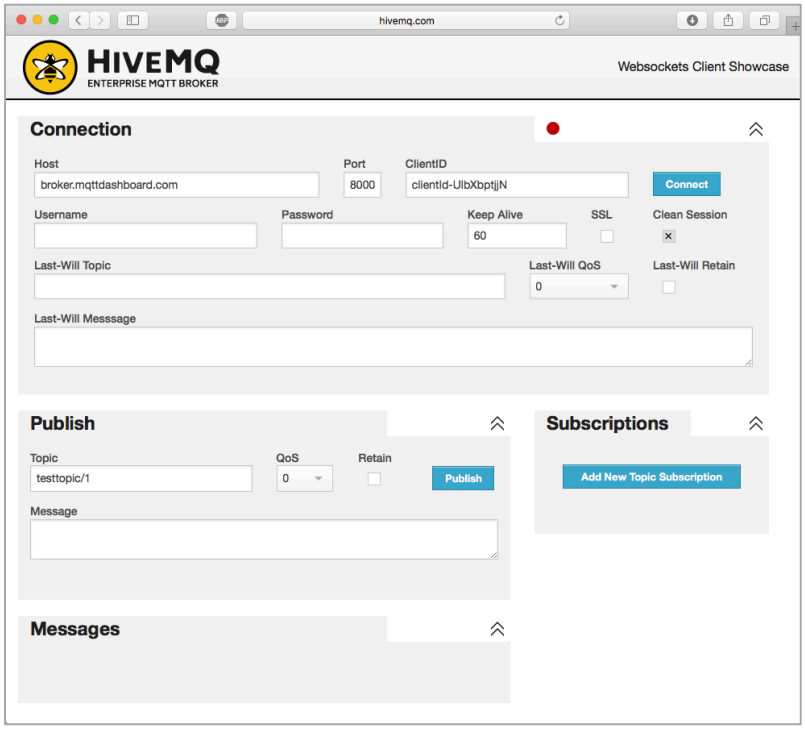



*Abbildung 77: HiveMQ Websocket Client*

- *mosquitto MQTT over Websockets* (Beispiel MQTT über Websockets): aufrufbar unter *test.mosquitto.org/ws.html* → CONNECT: ja, SUBSCRIBE: ja, PUBLISH: ja (nur beispielhaftes Testen einer vorgegebenen Verbindung)

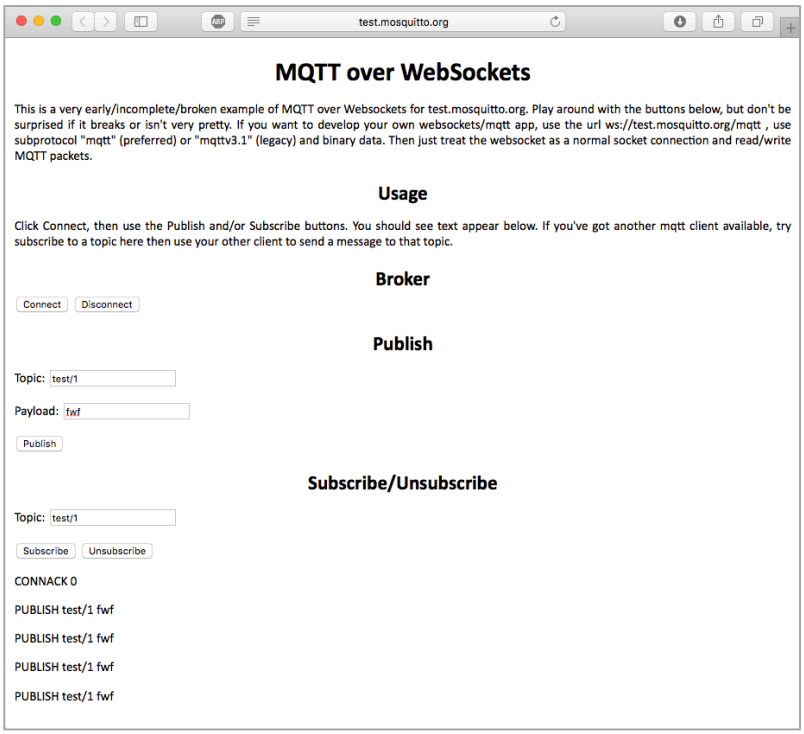



*Abbildung 78: MQTT over Websockets Example*

- *MQTT Client Example* (MQTT auf Websocket Beispiel): aufrufbar unter *mitsuruog.github.io/what-mqtt/* → CONNECT: ja, SUBSCRIBE: ja, PUBLISH: ja (nur Verbindung auf Websocket ohne Authentifizierung möglich, Beispielhaftes Verbinden, Subscriben und Publishen möglich)

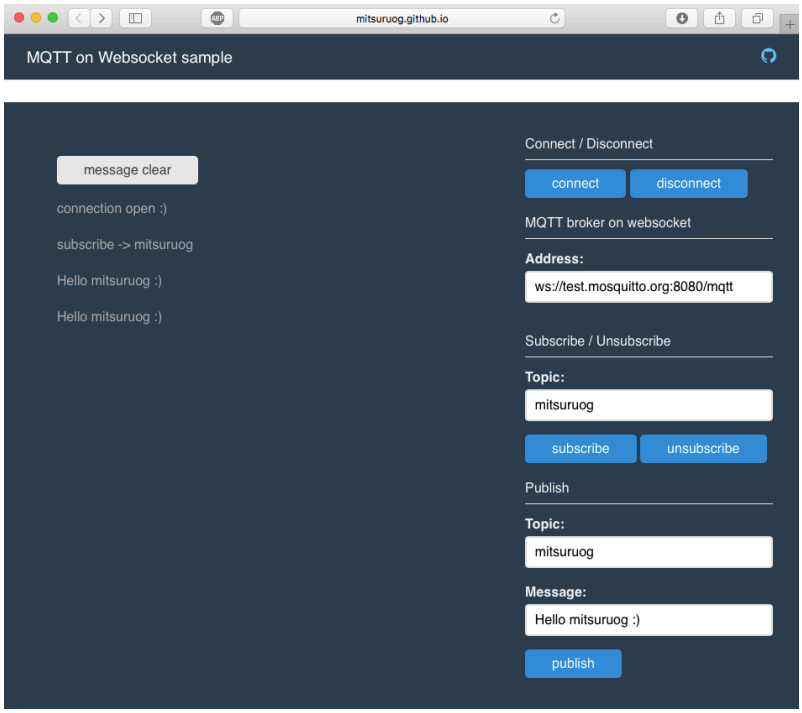



*Abbildung 79: MQTT Client Example*

- *ThingStudio* (Erstellen von HTML-Templates für User Interfaces, die MQTT-Daten visualisieren und Nachrichten über Steuerelemente senden können): Informationen unter *thingstud.io*

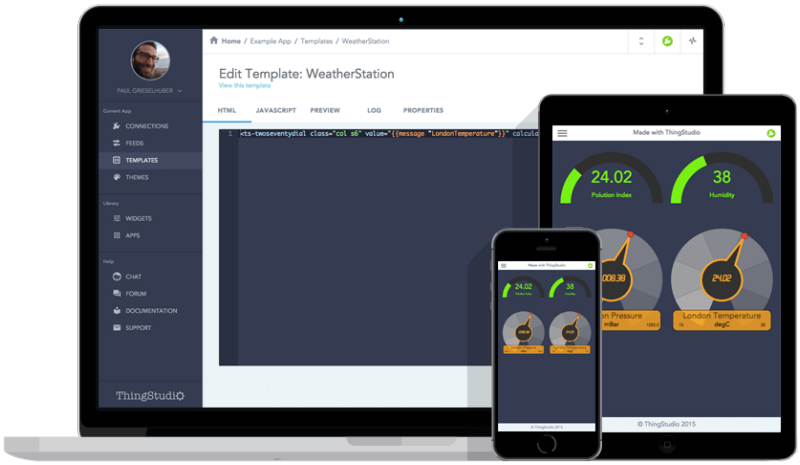



*Abbildung 80: ThingStudio Editor und User Interfaces*

- *MQTTlens* (Interface für MQTT-Verbindungen, verfügbar als Chrome Plugin): herunterzuladen unter *chrome.google.com/webstore/detail/mqttlens/hemojaaeigabkbcookmlgmdigohjobjm* → CONNECT: ja, SUBSCRIBE: ja, PUBLISH: ja

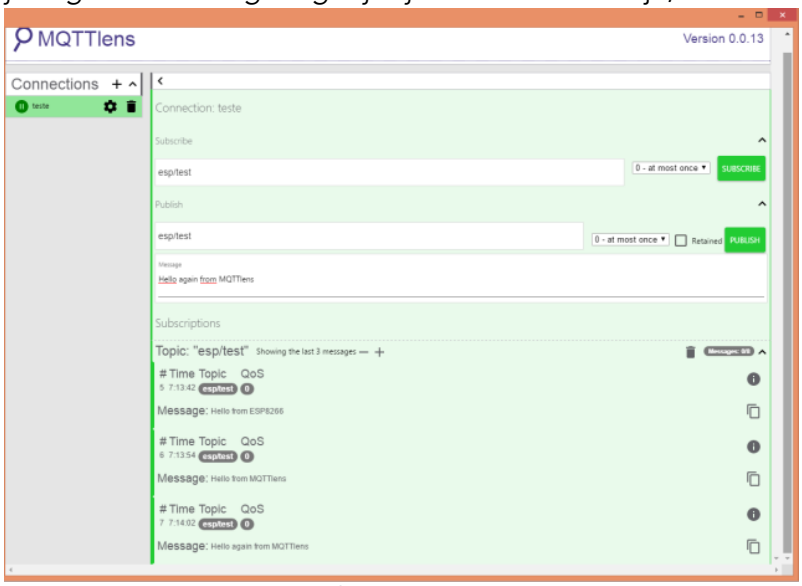



*Abbildung 81: MQTTlens Interface*

- *Paho Websocket Client* (Websockets Beispiel zum Testen): aufrufbar unter *eclipse.org/paho/clients/js/utility/* → CONNECT: ja, SUBSCRIBE: ja, PUBLISH: ja

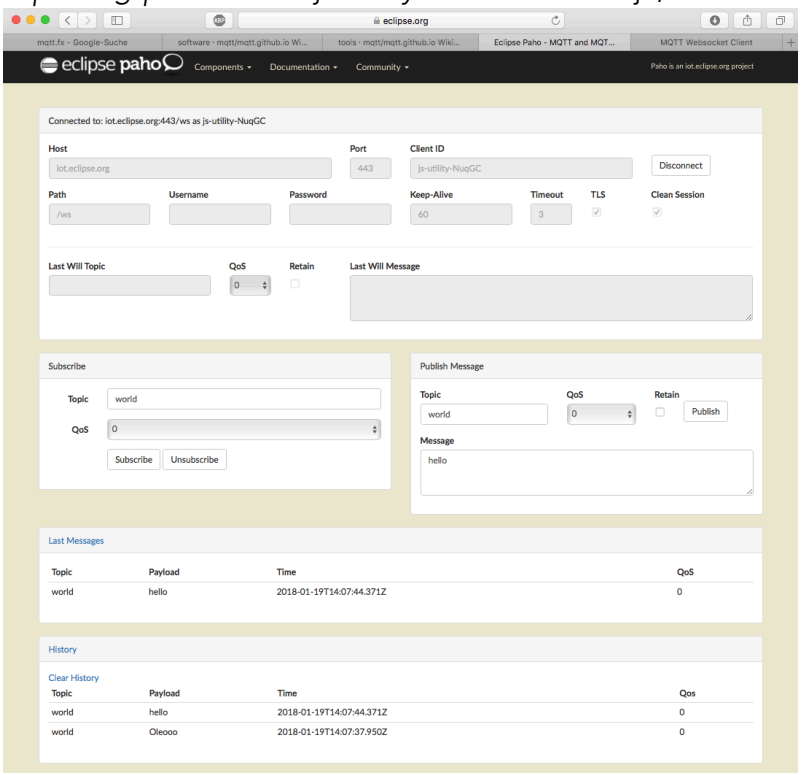



*Abbildung 82: Paho Websocket Client*

Bei der Auflistung von Tools für MQTT über Web-Browser wird ersichtlich, dass die Kommunikation von MQTT über Websockets eine Möglichkeit ist, Informationen schnell browserbasiert abrufen oder publizieren zu können. Das *MQTT Client Example* und der *Paho Websocket Client* funktionierten zuverlässig und können zum Testen dieser Kommunikationsmethode verwendet werden. Zusätzlich lassen sich einige der verbleibenden Tools dafür verwenden, Daten grafisch im Webbrowser aufzubereiten.

**Mobil:**

An dieser Stelle werden Apps vorgestellt, die die Kommunikation mit dem MQTT-Protokoll unterstützen. Untersucht werden Apps für iOS, die Programme für Android werden lediglich aufgelistet.

- *MQTTool* (iOS, kostenfrei): CONNECT: ja, SUBSCRIBE: ja, PUBLISH: ja

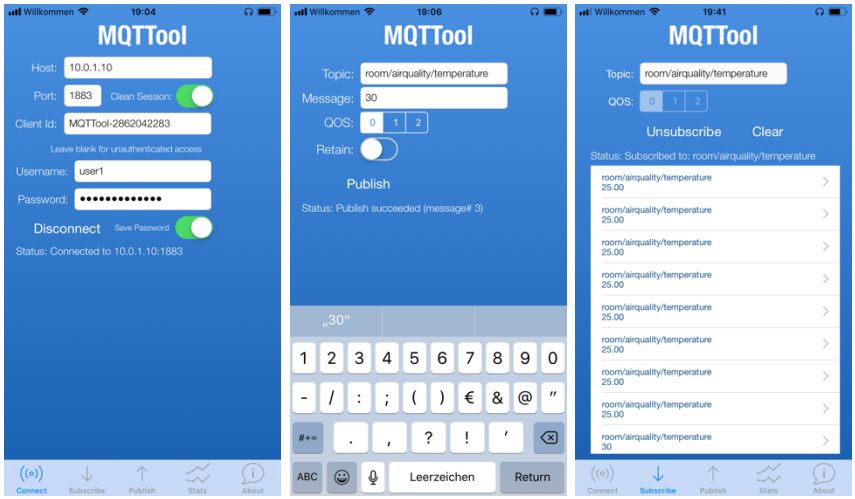



*Abbildung 83: MQTTool iOS App*

- *MQTTT* (iOS, kostenfrei): CONNECT: ja, SUBSCRIBE: ja, PUBLISH: ja

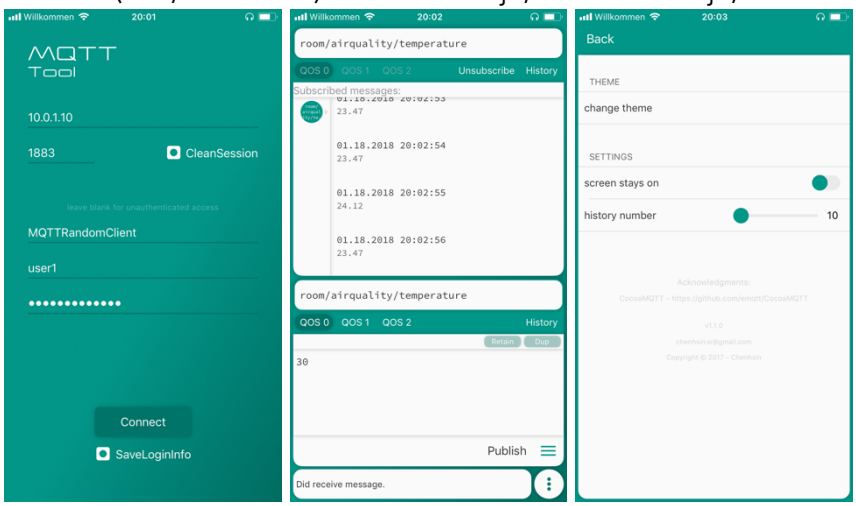



*Abbildung 84: MQTTT iOS App*

- *MQTT Tester* (iOS, 4,49 Euro): CONNECT: nein → keine Authentifikation möglich, SUBSCRIBE: nein, PUBLISH: nur bei InApp-Kauf

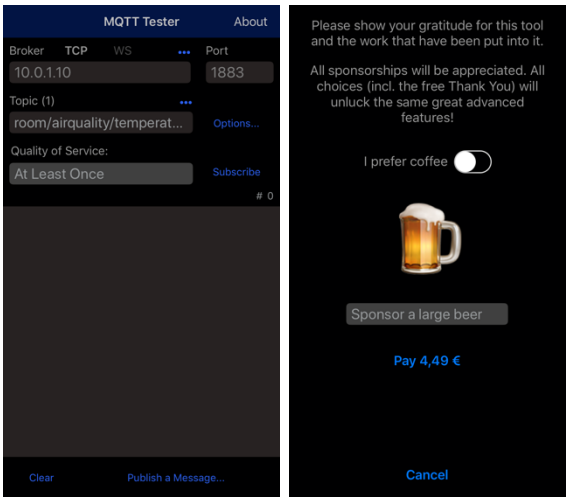



*Abbildung 85: MQTT Tester iOS App*

- *ICPDAS MQTT* (iOS, kostenfrei): CONNECT: ja, SUBSCRIBE: nein, PUBLISH: ja

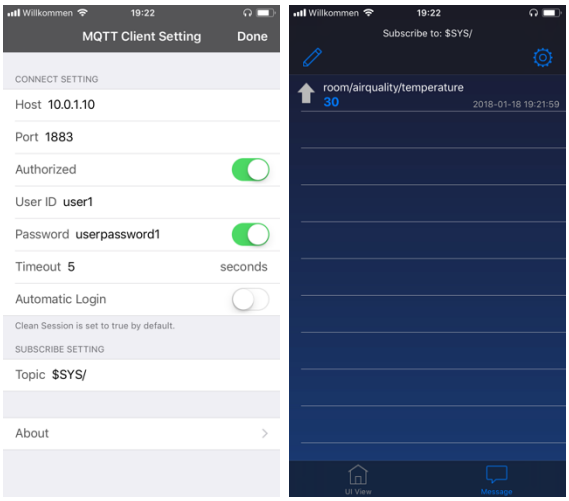



*Abbildung 86: ICPDAS MQTT iOS App*

- *Mqtt Buddy* (iOS, kostenfrei): CONNECT: ja, SUBSCRIBE: nein, PUBLISH: nein


*Abbildung 87: Mqtt Buddy iOS App*

- *MQTT Probe* (iOS, kostenfrei), CONNECT: nein → kein Verbindungsaufbau möglich, SUBSCRIBE: nein, PUBLISH: nein

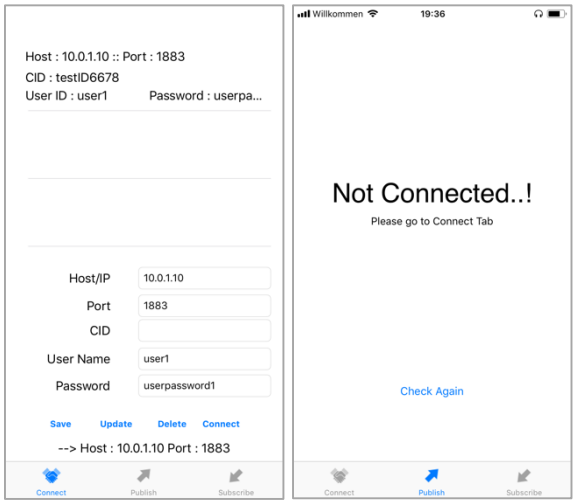



*Abbildung 88: MQTT Probe iOS App*

- *mqttclient* (iOS, kostenfrei): CONNECT: ja, SUBSCRIBE: nein, PUBLISH: nein

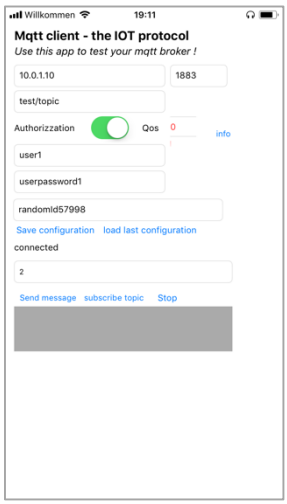



*Abbildung 89: mqttclient iOS App*

Folgende Apps sind zusätzlich verfügbar, wurden aber nicht getestet:

- *MQTTInspektor* (iOS, 2,29 Euro)
- *MQTT Manager* (iOS, 8,99 Euro)
- *MyMQTT* (Android, kostenlos)
- *MQTT Client* (Android, kostenlos)
- *IoT MQTT Dashboard* (Android, kostenlos)
- *MQTT Snooper* (Android, kostenlos)
- *MQTT Client for EasyControl* (Android, kostenlos)
- *Linear MQTT Dashboard* (Android, kostenlos)
- *MQTT Patterns* (Android, kostenlos)

Es zeigt sich, dass es im Google Playstore eine größere Anzahl an Android-Apps für MQTT gibt, als im App Store für iOS. Es gibt für iOS zwar einige kostenlose Apps, jedoch erscheinen die meisten davon qualitativ minderwertig. Zwei Ausnahmen machen die Apps *MQTTTool* und *MQTTT* für iOS. Diese stechen durch einfache Bedienung hervor und funktionieren wie erwartet.

# 4. Implementierung und Dokumentation eines MQTT-Testbettes

## 4.1. Grundlegende Anforderungen

Im praktischen Teil dieser Arbeit soll anhand eines MQTT-Testbettes untersucht werden, wie das MQTT Protokoll implementiert werden kann und wie gut es sich für eine konkrete Anwendung im Internet of Things eignet. Das Testbett soll später von Studenten in einem Labor-Versuch an der Hochschule Offenburg nachgebaut und ausprobiert werden können.

**Anwendung 1**
Den Studenten wird eine fertig implementierte Anwendung bereitgestellt. Die technischen Anforderungen dazu lauten:

I. Ein *Broker (Raspberry Pi) verwaltet* die *Kommunikation* zwischen den Clients
II. Ein *publizierender Client (ESP32)* soll Sensordaten *veröffentlichen*
III. Ein *ESP32 Client subskribiert* sich beim Broker und führt daraufhin *Aktionen* aus
IV. Ein weiterer *subskribierender Client (mobile Web-App) visualisiert* die Daten
V. Ein *Platform as a Service* Dienst wird in die Anwendung mit eingebunden und *stellt* die anfallenden *Informationen dar*

**Anwendung 2**
Die Studenten können sich an der ersten Anwendung orientieren und werden unter Zuhilfenahme der in dieser Arbeit erstellten Dokumentation das Testbett um folgende Komponenten erweitern:

VI. Ein *publizierender Client (ESP32)* soll Daten eines anderen Sensors *veröffentlichen*
VII. Ein zusätzlicher *subskribierender Client (ESP32)* reagiert auf bestimmte Werte des Sensors und führt daraufhin eine *Aktion* aus
VIII. Die vorhandene *mobile App* (subskribierender Client) wird um die Visualisierung der Daten des zweiten Sensors *erweitert*
IX. Die Informationen des Sensors sollen ebenfalls an den *PaaS-Dienst* gesendet werden

## 4.2. Anwendungsspezifische Anforderungen

**Anwendung 1: UV-Index Detektor**
Nicht nur im Sommer setzen wir uns täglich einer sichtbaren Gefahr aus: Von dem menschlichen Auge nicht wahrnehmbare UV-Strahlen schaden Haut und Augen. Bei längerem und wiederholtem Aufenthalt in der Sonne ist ein hohes Krebsrisiko für sämtliche Hauttypen nicht von der Hand zu weisen. Gerade Kinder und Jugendliche sind davon betroffen. Mit einem wirksamen Sonnenschutz lässt sich gegen eine solche Gefahr angehen. Doch den meisten Menschen ist die unmittelbare Gefährdung nicht bewusst.

Zwar warnt ein durch Meteorologen regional vorhergesagter UV-Index im Internet bereits vor zu hohen Werten [68], dieser muss jedoch durch den Nutzer aktiv über das Internet abgefragt werden.

Bei dem Testbett ist vorgesehen, dass eine außen angebrachte lokale Messstation für UV-Strahlung Messwerte liefert, die beim User auf dem Smartphone angezeigt oder im Haus anhand einer Ampel-Anzeige visualisiert werden. In diesem Fall würden die Erziehungsberechtigten ihren Kindern je nach Intensität der UV-Sonnenstrahlung den richtigen Sonnenschutz bereitstellen oder den Aufenthalt in der Sonne untersagen können. Die Anforderungen zu der Messstation zum UV-Index sollen folgendes beinhalten:

I. Die *Mosquitto-Implementierung* des *MQTT-Brokers*, die auf dem Raspberry Pi installiert ist, *verwaltet* die *Kommunikation* zwischen allen Clients
II. Ein UV-Sensor, angeschlossen an den ESP32, detektiert den aktuellen UV-Index und *veröffentlicht* diesen an den Broker
III. Ein *ESP32 Client subskribiert* sich beim Broker und bekommt die Werte weitergeleitet. Eine daran angeschlossene *NeoPixel-Anzeige* gibt je nach *UV-Index verschiedene Farben* aus, die über die aktuell vorherrschende Strahlung informieren sollen
IV. Eine *mobile Web-App* soll *visuell* darstellen, was der gemessene Wert ausdrückt. Die App wird über das *Node-RED Dashboard* realisiert und ist ein subskribierender Client
V. Um eine Historie über die gemessenen Wert überblicken zu können, werden die Daten über MQTT an den *PaaS-Dienst Adafruit IO* gesendet und dort gespeichert

Anwendung 2: Luftqualität Messstation

Ein anderes Problem stellt schlechte oder verbrauchte Luft in Innenräumen dar. Beispielsweise in Schulen, wo in Klassenräumen eine schlechte Luftqualität für effektive Lernprozesse hinderlich sein kann. Hier wird eine Visualisierung der aktuellen Luftqualität aufzeigen, dass der überwachte Raum einer ordentlichen Durchlüftung bedarf. Dafür sind folgende Anforderungen vorgesehen:

VI. Ein *Luftqualität-Sensor* liefert Messwerte an einen *ESP32 Mikrocontroller*, der die Daten *veröffentlicht*
VII. Ein weiterer *ESP32 subskribiert* sich beim Broker. Wenn die Messung einen definierten Wert übersteigt, wird ein an den ESP32 angeschlossener *Servomotor* in Gang gesetzt, der einen *Fenster-Motor* repräsentieren soll. Dieser soll die *Öffnung und Schließung des Fensters* darstellen
VIII. Die *Node-RED App* wird um ein weiteres *Anzeige-Element erweitert*, das die aktuelle Luftqualität darstellt
IX. Auch hier wird die Anbindung zu dem *PaaS-Dienst Adafruit IO* realisiert, indem die Werte an die Plattform veröffentlicht werden

## 4.3. Konzept

### 4.3.1. Sequenzdiagramme

**Anwendung 1: UV-Index Detektor**

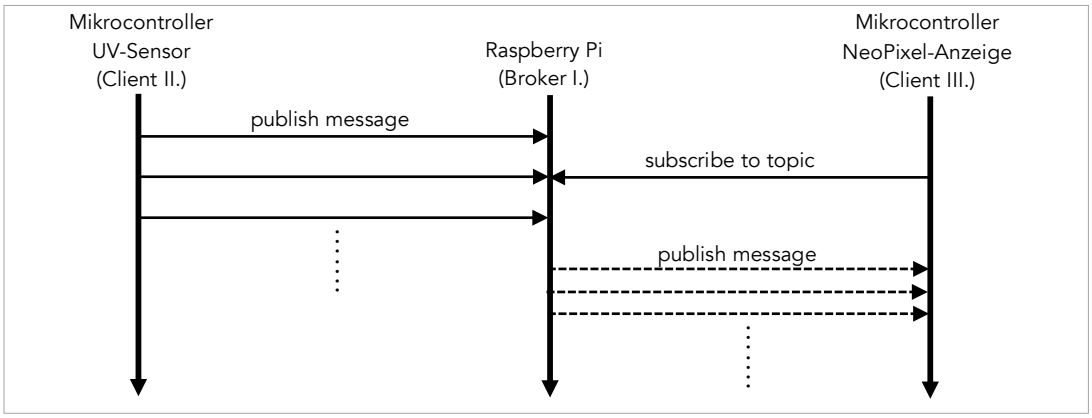



*Abbildung 90: Sequenzdiagramm NeoPixel Client*

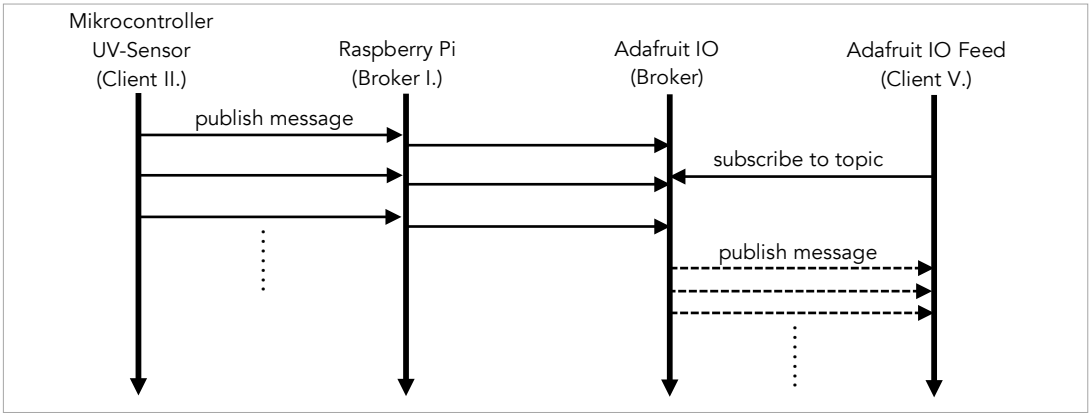



*Abbildung 91: Sequenzdiagramm PaaS-Dienst Client*

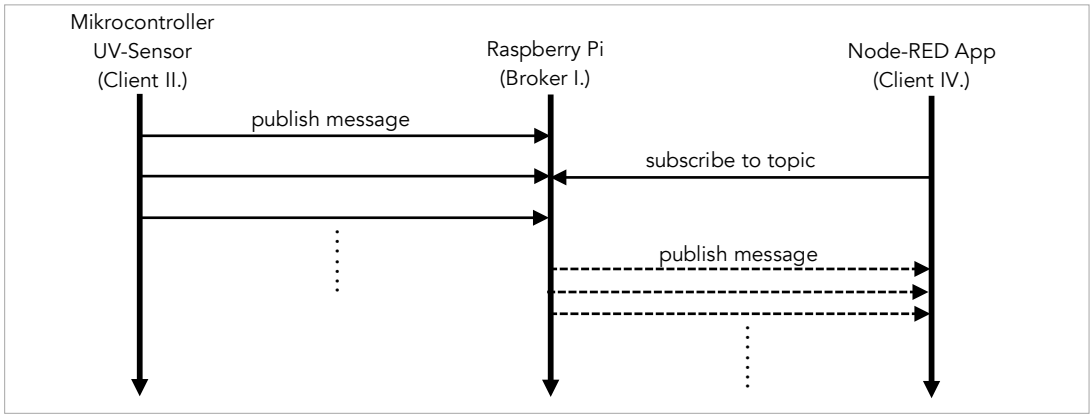



*Abbildung 92: Sequenzdiagramm Node-RED App Client*

## Anwendung 2: Luftqualität Messstation

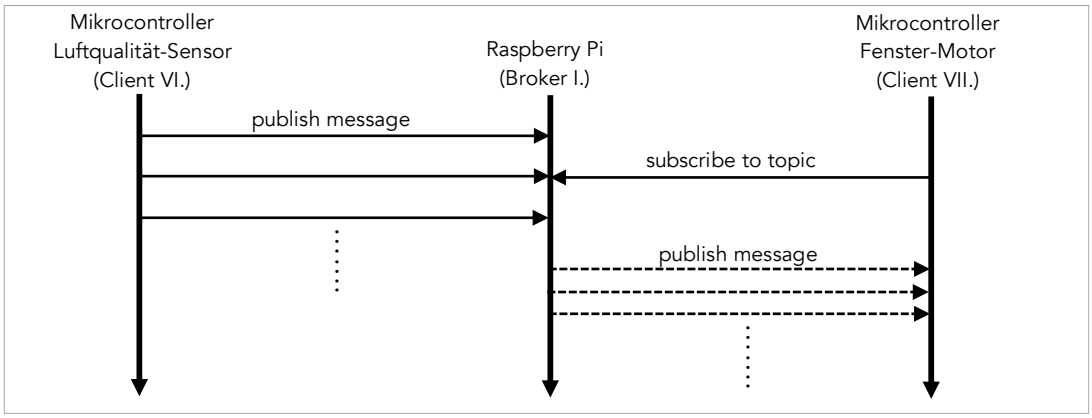



*Abbildung 93: Sequenzdiagramm Fenster-Motor Client*

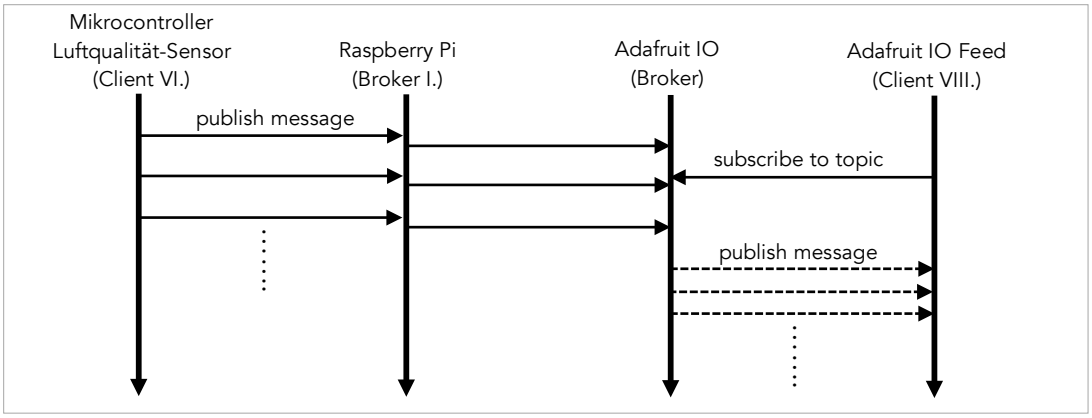



*Abbildung 94: Sequenzdiagramm PaaS-Dienst Client*

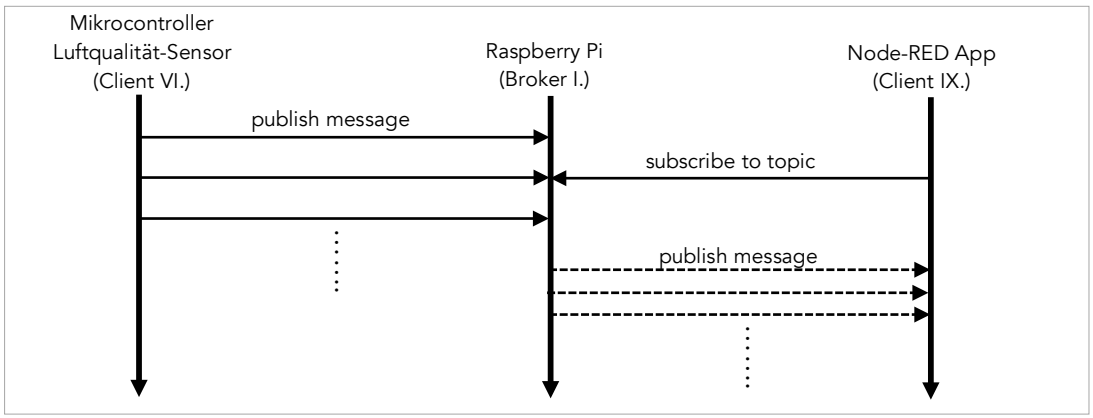



*Abbildung 95: Sequenzdiagramm Node-RED App Client*

### 4.3.2. Gesamtarchitektur der Anwendung

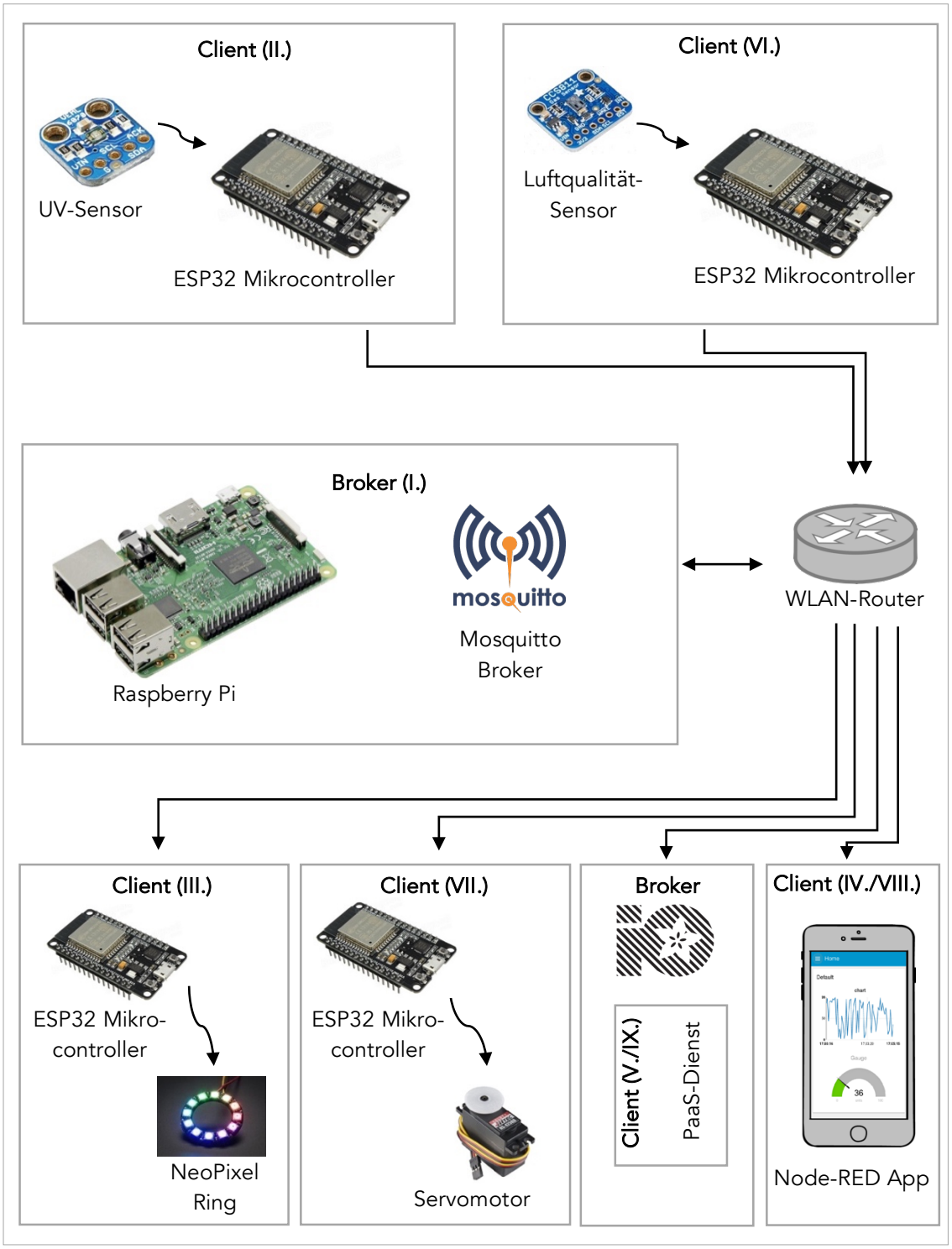



*Abbildung 96: Architektur des Systems*

### 4.3.3. Liste der Hardware

#### Raspberry Pi 3 Model B

*Abbildung 97: Raspberry Pi*

Der Raspberry Pi ist ein Einplatinencomputer mit einem ARM-Prozessor. Er hat unterschiedliche Schnittstellen und kann über digitale Ein- und Ausgänge um Sensoren und Ähnliches erweitert werden. Die auf einer Mikro-SD-Karte installierte Linux-Distribution oder andere Betriebssysteme ermöglichen den Umgang anhand einer visuellen Benutzeroberfläche. Die hier verwendete Distribution nennt sich Raspbian Stretch und ist das am weitesten verbreitete Betriebssystem für den Raspberry. Ab der Version des Raspberry Pi 3 Model B, die hier Verwendung findet, sind Wi-Fi und Bluetooth standardmäßig eingebaut.

#### ESP32 Development Board

*Abbildung 98: ESP32 Development Board*

Der ESP32-Chip ist der Nachfolger des bekannten ESP8266 Mikrocontrollers. Er unterstützt WLAN-Konnektivität und hat einen niedrigen Energiebedarf. Eine Neuerung gegenüber des Vorgängermodells ist die nun vorhandene Bluetooth 4.2 BLE Konnektivität. Hergestellt wird der ESP32 von der Firma Espressif. Weitere integrierte Interfaces, die für IoT von Bedeutung sind, sind beispielsweise mehrere Touch-Sensoren (über kapazitive GPIO Überwachung), ein Temperatursensor, ein Infrarot Remote Controller u.v.m. Programmiert wird der ESP32 über die Arduino IDE.

#### Servomotor

*Abbildung 99: Servomotor*

Servomotoren lassen sich individuell über Winkel steuern. Es muss nur der Rotationswinkel angegeben werden. Die Signale werden über Pulsweitenmodulation über die GPIO Pins des Mikrocontrollers an den Motor gesendet.

#### Adafruit VEML6070 UV Sensor Breakout


*Abbildung 100: Adafruit VEML6070 UV Sensor Breakout*

Mit dem Adafruit VEML6070 UV-Sensor lässt sich die Lichtintensität im Spektrum von UV-Strahlen messen. Dies geschieht über die I2C-Schnittstelle des Mikrocontrollers. Als Sensor-Chip wird der VEML6070 Sensor der Firma Vishay verwendet. Als Wert wird nicht wie bei anderen UV-Sensoren der offizielle UV-Index zurückgegeben, sondern es handelt sich um die Menge an Sonneneinstrahlung im Verhältnis Leistung pro Fläche.

### Adafruit CCS811 Luftqualität-Sensor VOC Breakout


*Abbildung 101: Adafruit CCS811 Luftqualität-Sensor VOC Breakout*

Der Adafruit CCS811 Air Quality Sensor Breakout beinhaltet den CSS811 Gas Sensor von AMS und ist für die Überwachung von Luftqualität in Innenräumen geeignet. Dieser Sensor kann flüchtige organische Verbindungen (VOCs), also flüchtige Gase in der Umgebungsluft erkennen. Dieser Breakout ist bei der Verwendung der Adafruit Library in der Lage, über die I2C Schnittstelle des Mikrocontrollers neben einem totalen Wert für VOC (TVOC) auch einen Wert für das $CO_2$-Äquivalent zu liefern (eCO2). Zusätzlich bietet der Sensor die Möglichkeit, die aktuelle Umgebungstemperatur zu messen.

### Adafruit NeoPixel Ring – 12x RGB LED

Ein NeoPixel Ring ist eine kreisrunde Leiterplatte, die mit NeoPixel LEDs versehen ist. Es kann zischen einer unterschiedlichen Anzahl an Pixeln gewählt werden. In dieser Arbeit wird ein Ring mit zwölf LEDs verwendet. Alle LEDs können einzeln über den integrierten Treiber und die dazugehörige Adafruit Bibliothek angesprochen werden.

*Abbildung 102: Adafruit NeoPixel Ring*

### 4.3.4. Verwendete Software

#### MQTT.fx

*MQTT.fx* ist eine Implementierung eines MQTT-Clients für Plattformen wie Windows, MacOS und Linux. Es dient als freies Werkzeug zum Testen und Debuggen von MQTT-Kommunikation und ist unter der Apache License 2.0 veröffentlicht. Die Kommunikation kann ebenfalls über die Kommandozeile durch Befehle getestet werden. Das auf JavaFX und Eclipse Paho aufbauende Tool bietet jedoch über eine grafische Oberfläche sämtliche Funktionalitäten von MQTT-Features. Auf einfachem Wege kann eine Kommunikation mit dem Broker hergestellt werden. Zu den einzelnen Funktionen zählen beispielsweise das Anlegen von Verbindungsprofilen für unterschiedliche Broker-Verbindungen, MQTT-Security, das Publizieren und Subskribieren von Topics und das Ausführen von vordefinierten Skripts.

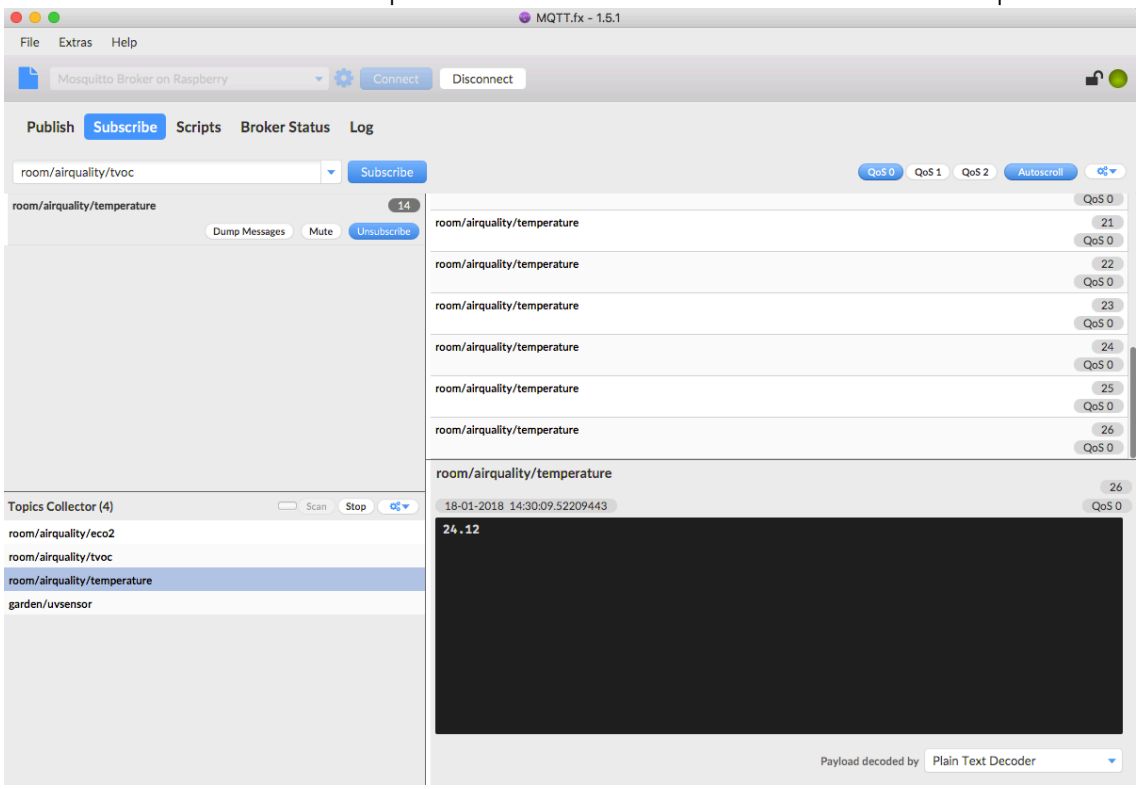



*Abbildung 103: MQTT.fx Benutzeroberfläche*

#### Arduino IDE

Die Java-basierte *Arduino IDE* ist eine frei verfügbare integrierte Entwicklungsumgebung für Mikrocontroller. Integriert ist ein Code-Editor mit *Syntax-Highlighting* und der *GCC-Compiler*. In der IDE werden sogenannte *Sketches* erstellt, die mithilfe von unterschiedlichen Libraries in einer *C* und *C++* ähnlichen Programmiersprache geschrieben werden. Anschließend kann ein Sketch einfach kompiliert und auf den Mikrocontroller hochgeladen werden. Ein Arduino-Sketch benötigt immer zwei Methoden: Die beim Programmstart einmalig ausgeführte *setup()*-Funktion wird dazu verwendet, Variablen zu initialisieren, Input- oder Output-Pins und Libraries zu definieren. Wiederholend wird anschließend die *loop()*-Methode ausgeführt, in der die definierten Programmfunktionen und die Logik stattfinden.

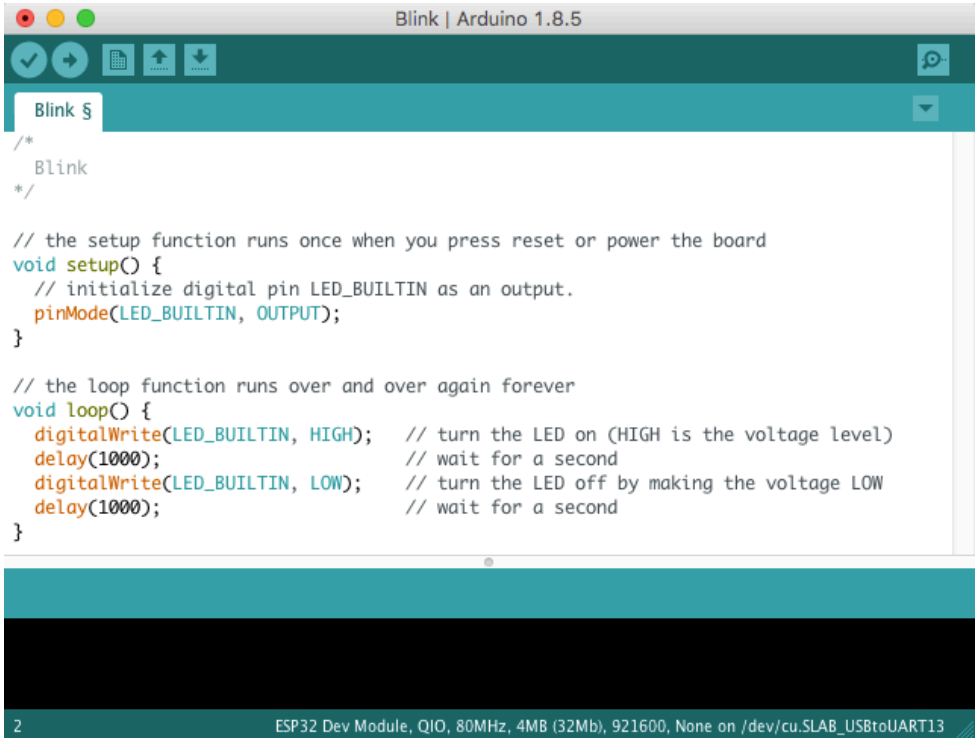



*Abbildung 104: Arduino IDE Code-Editor*

## Node-RED

*Node-RED* ist eine visuelle Programmierumgebung, mit der eine Verknüpfung von Hardware, Online-Services und unterschiedlicher Programmierschnittstellen möglich ist. Das Programm ist ein Projekt der *JS Foundation und* kostenfrei nutzbar. Es ist in erster Linie beim Umsetzen weniger komplexer Projekte geeignet. Node-RED wird dabei von allen gängigen Betriebssystemen unterstützt. Fertige Module, die auch nachträglich installiert werden können, beziehungsweise deren Funktionen (sogenannte *Nodes*), werden als Blöcke dargestellt. Miteinander verbunden werden sie *Flow* genannt. Gestartet wird das Programm über die Konsole, die Programmier-Oberfläche wird im Browser aufgerufen.

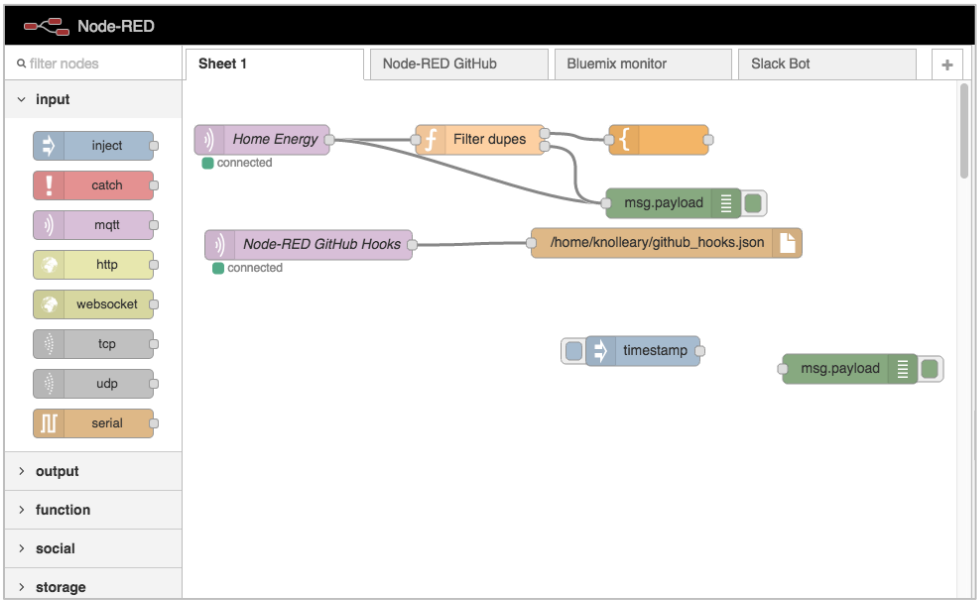



*Abbildung 105: Node-RED Oberfläche*

## 4.4. Realisierung und Dokumentation

### 4.4.1. Anwendung 1: UV-Index Detektor

#### I. Installation des MQTT Mosquitto Broker auf dem Raspberry Pi

Das normale *apt-get* Archiv beinhaltet nicht die aktuelle Version von Mosquitto. Aus diesem Grund wird vorher am Terminal der Signaturschlüssel für das aktuelle Repository Package geholt:

```
$ wget http://repo.mosquitto.org/debian/mosquitto-repo.gpg.key
$ sudo apt-key add mosquitto-repo.gpg.key
```

Dann wird das Repository geladen und installiert:

```
$ cd /etc/apt/sources.list.d/
$ sudo wget http://repo.mosquitto.org/debian/mosquitto-stretch.list
$ sudo apt-get update
```

Als nächstes werden der *Mosquitto Broker*, der Mosquitto Client und die zugehörige Python-Library installiert:

```
$ sudo apt-get install mosquitto mosquitto-clients python-mosquitto
```

Der Server wird nach der Installation automatisch gestartet. Durch den folgenden Befehl lässt sich der Server stoppen:

```
$ sudo /etc/init.d/mosquitto stop
```

Auf die Konfigurationsdatei des Brokers kann über diesen Befehl zugegriffen werden. In dieser soll jedoch keine Veränderung vorgenommen werden:

```
$ sudo nano /mosquitto/mosquitto.conf
```

Zum Erstellen einer lokalen Konfigurationsdatei wird in folgendem Verzeichnis eine neue Datei *default.conf* erstellt:

```
$ sudo nano /etc/mosquitto/conf.d/default.conf
```

In dieser werden folgende Einträge vorgenommen:

```
# Local default mosquitto conf file

allow_anonymous false
password_file /etc/mosquitto/passwd
```

Die zwei Einträge in der Datei bewirken, dass nicht authentifizierte Verbindungen verhindert werden und die angegebene Datei *passwd* einen User mit dem dazugehörigen Passwort speichert. Um einen neuen Benutzer festzulegen, muss diese Anweisung ausgeführt werden:

```
$ sudo mosquitto_passwd -c /etc/mosquitto/passwd user1
```

Daraufhin kann das zu dem Nutzer *user1* zugehörige Passwort zweimal eingegeben werden. Auf diese Änderung hin muss das Gerät, beziehungsweise der Dienst neu gestartet werden.

Anschließend kann der Mosquitto Broker gestartet werden, er ist aber ursprünglich so eingestellt, dass er sich nach dem Systemstart automatisch startet:

```
$ sudo /etc/init.d/mosquitto start
```

Nachdem der Dienst mit folgender Meldung den Start des Servers meldet, kann mit dem MQTT.fx Tool die Server-Funktionalität getestet werden:

```
[ ok ] Starting mosquitto (via systemctl): mosquitto.service
```

Für den kommenden Schritt wird die IP-Adresse des Gerätes erforderlich, auf dem der Broker läuft. Sie lässt sich auf selbigem über den Befehl herausfinden:

```
$ ifconfig
```

### Testen der Server-Funktionalität auf einem beliebigen Client

Auf einem Device, auf dem das Java-Programm MQTT.fx läuft, wird anschließend unter dem Reiter *Extras -> Edit Connection Profiles* ein neues Profil für einen Client erstellt, indem unten links auf das blaue *Plus* geklickt wird. Ein Profilname, die notierte IP-Adresse und die Portnummer des Brokers werden nun eingegeben:

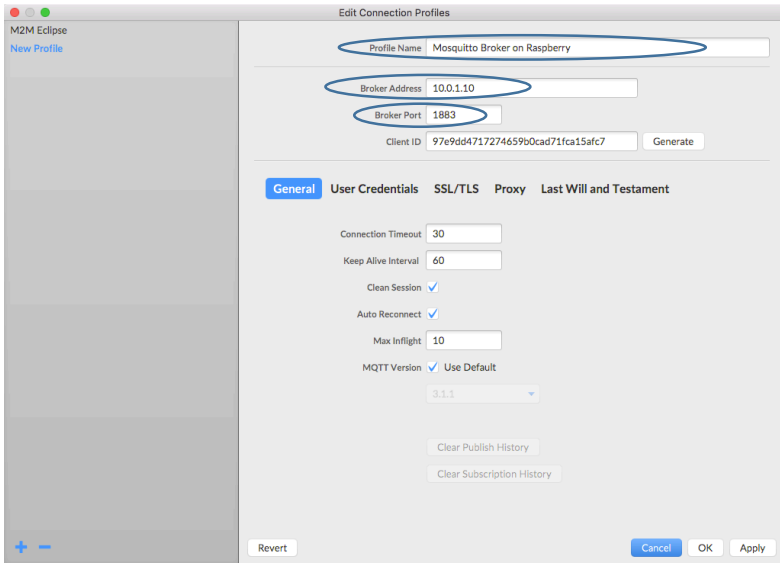



*Abbildung 106: MQTT.fx Einrichtung eines Profils*

Weiterhin müssen die oben definierten Credentials unter dem Reiter *User Credentials* eingegeben werden:

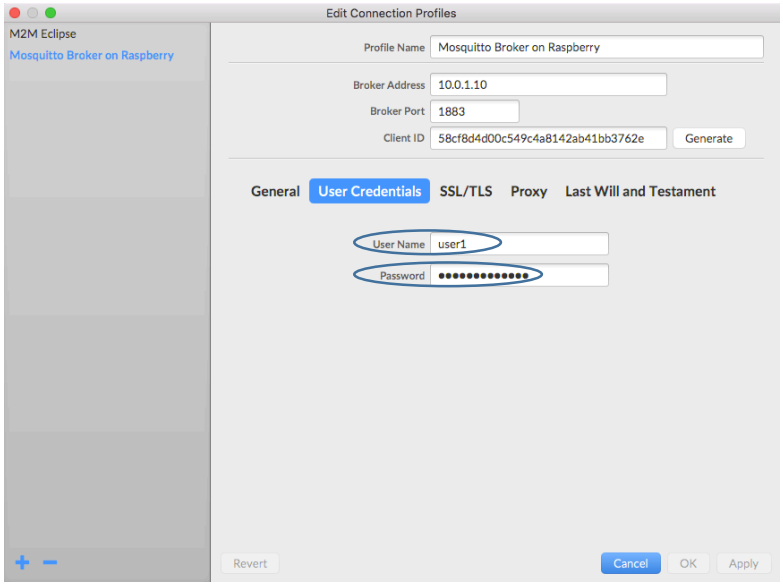



*Abbildung 107: MQTT.fx Einrichtung eines Profils (2)*

Mit diesem Schritt wird ein Profil für einen Client erstellt. Durch den Klick auf *Connect* auf der Oberfläche wird eine Verbindung zum Broker hergestellt:

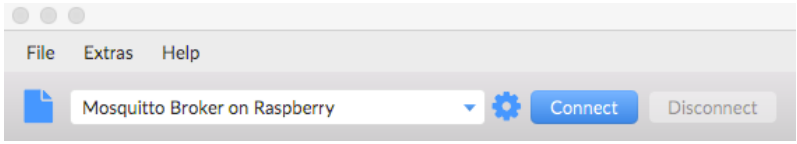



*Abbildung 108: MQTT.fx Connect*

Wenn die Verbindung erfolgreich hergestellt worden ist, wird rechts oben ein grüner Punkt angezeigt und es kann unter der Schaltfläche *Publish* ein Topic Name und der Payload eingegeben werden. Nach einem Klick auf den *Publish Button* neben dem Topic Eingabefeld wird der eingegebene Wert *23* testweise als Payload unter dem Topic Name `haus/badezimmer/temperatur` an den Broker publiziert. Damit der Wert beim Vorgang des Subskribierens eines neuen Clients diesem als letzter Wert angezeigt wird, wird das *Retained Flag* durch die entsprechende Schaltfläche aktiviert.

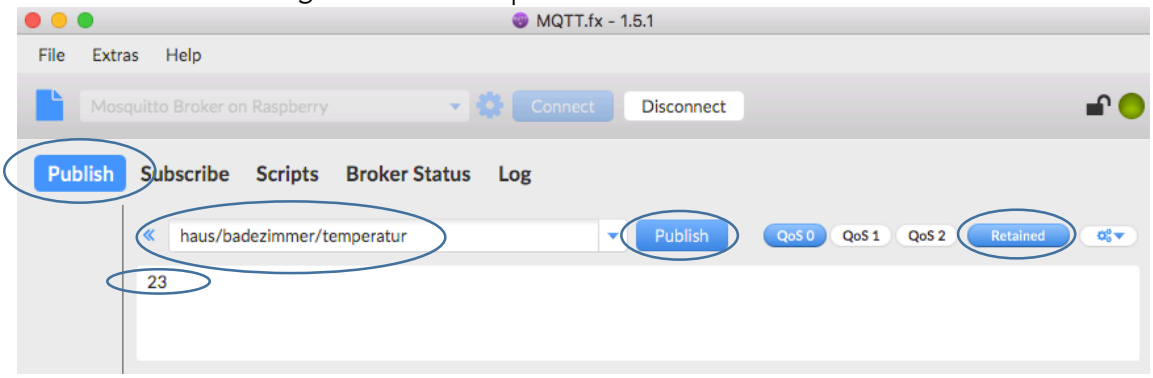



*Abbildung 109: MQTT.fx Publish*

Um zu testen, ob das Publishen an den Broker funktioniert hat, wird die Funktionalität des Clients unter der Schaltfläche auf *Subscribe* gestellt. Nach der Eingabe des oben genannten Topic Name in das Eingabefeld und dem Klick auf den *Subscribe Button* wird der zuletzt publizierte Wert im Feld des Payloads angezeigt.

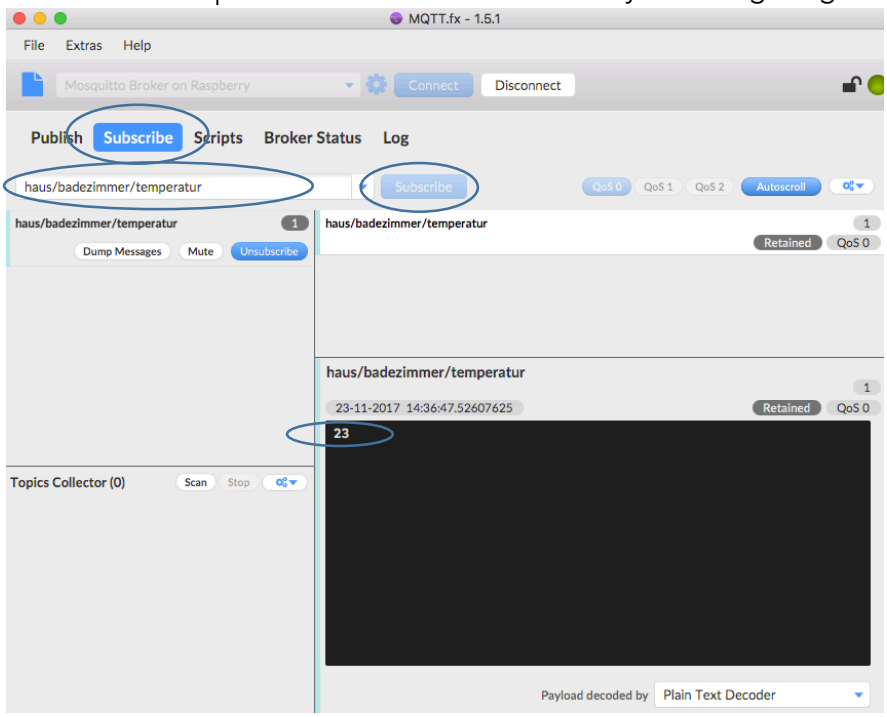



*Abbildung 110: MQTT.fx Subscribe*

Der Verbindungsaufbau zum Broker, das Publishen und das Subskribieren wurden somit erfolgreich getestet. Unter der Schaltfläche *Broker Status* kann sich der Client zu den unterschiedlichen vom Server bereitgestellten Status Topics subskribieren, um Systeminformationen über den Broker, Clients und Nachrichten zu erhalten.

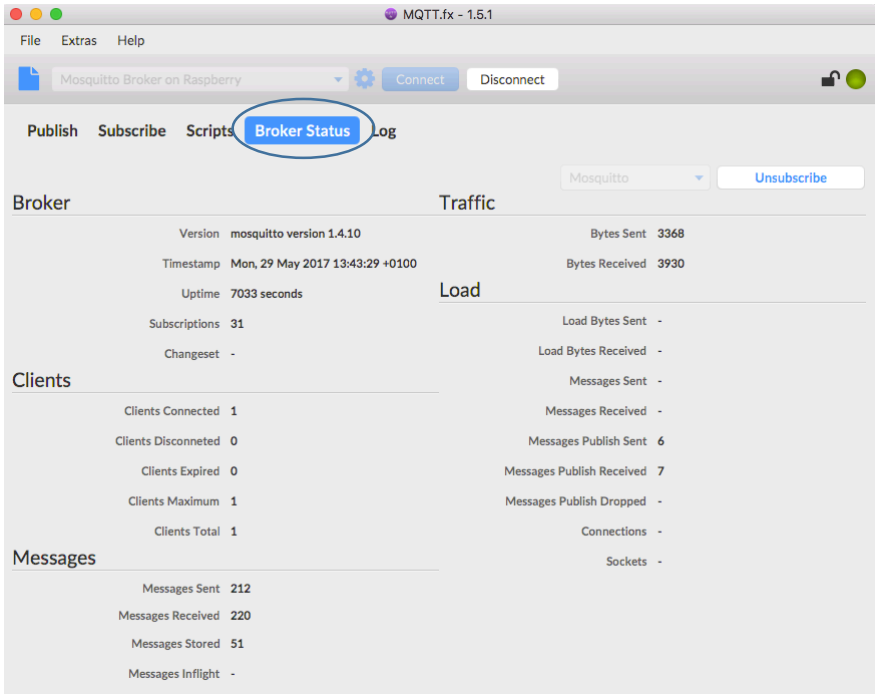



*Abbildung 111: MQTT.fx Subscribe Broker Status Variablen*

### II. Einrichten des Clients auf dem ESP32 mit UV-Sensor

Der ESP32 Mikrocontroller wird mithilfe der Arduino IDE programmiert. Vor der Arbeit mit dem Mikrocontroller ESP32 muss eine Bibliothek für die Programmierung des Boards unter der Umgebung, in der die Arduino IDE läuft (in diesem Fall Mac), eingebunden werden. Für diesen Schritt muss das Arduino ESP32 Git Repository geklont und installiert werden. Ausführliche Anleitungen für andere Plattformen gibt es unter folgendem Link: *github.com/espressif/arduino-esp32*

```
$ mkdir -p ~/Documents/Arduino/hardware/espressif
$ cd ~/Documents/Arduino/hardware/espressif
$ git clone https://github.com/espressif/arduino-esp32.git esp32
$ cd esp32
$ git submodule update --init --recursive
$ cd tools
$ python get.py
```

Um zu überprüfen, ob die Bibliothek für das Board installiert wurde, muss nun in der Arduino IDE unter dem Reiter *Tools -> Boards* das *ESP32 Dev Module* auswählbar sein:

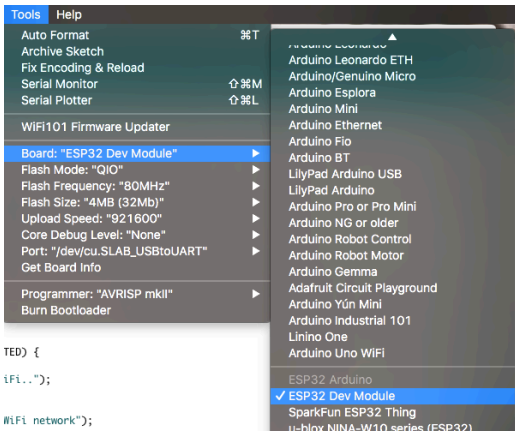



*Abbildung 112: Arduino IDE Auswahl des Boards*

Weiterhin ist es unter Mac erforderlich, sogenannte *UART-Treiber* zu installieren. Diese machen es möglich, eine serielle Schnittstelle über USB verfügbar zu machen. Dafür legt der Treiber einen virtuellen *COM Port (VCP)* an. Ein solcher VCP Treiber ist unter der Adresse für alle gängigen Betriebssysteme herunterzuladen:
*silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers*
Nachdem die ausführbare Datei den Treiber installiert hat, ist in der Arduino IDE zu prüfen, ob die Installation erfolgreich war:

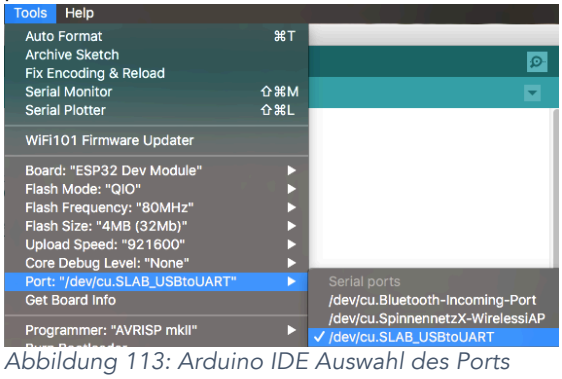



*Abbildung 113: Arduino IDE Auswahl des Ports*

Im kommenden Schritt wird geprüft, ob der ESP32 Mikrocontroller richtig eingerichtet wurde. Dafür wird ein einfacher Beispiel-Code verwendet, der den ESP32 jegliche in der Umgebung befindlichen WLAN-Netzwerke scannen lässt und deren SSIDs und die jeweilige Empfangsstärke im seriellen Monitor anzeigt. Unter der Schaltfläche *File* ->

*Examples (Examples for ESP32 Dev Module) -> WiFi* findet sich das Code-Beispiel *WiFiScan*. Unter *Tools* kann der *Serial Monitor* geöffnet werden, um den Output des Test-Programms auszugeben. Bei dem Serial Monitor muss für dieses Programm die *Baudrate* auf den Wert *115200* eingestellt werden, um einen verwertbaren Output zu bekommen (siehe Abbildung 114). Die Ausgabe sollte dann etwa folgendermaßen aussehen:

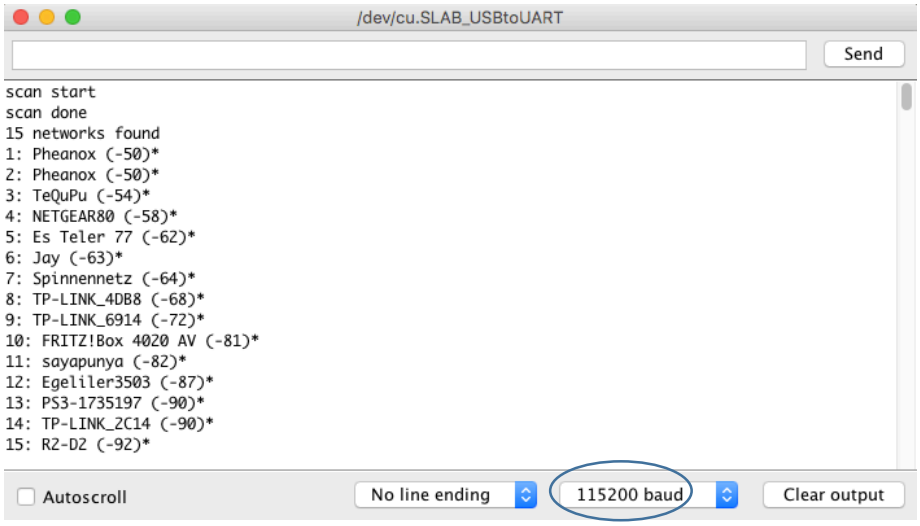



*Abbildung 114: Arduino IDE Serial Monitor*

Um den ESP32 als MQTT Client verfügbar zu machen und ihn mit dem Broker zu verbinden, wird eine zusätzliche Library benötigt. Hier wird die *PubSubClient*-Library verwendet. Dafür wird im *Library Manager* nach der gewünschten Bibliothek gesucht und diese direkt installiert. Die Verwaltung der Bibliotheken ist über die Schaltfläche *Sketch -> Include Library -> Manage Libraries* zu erreichen:

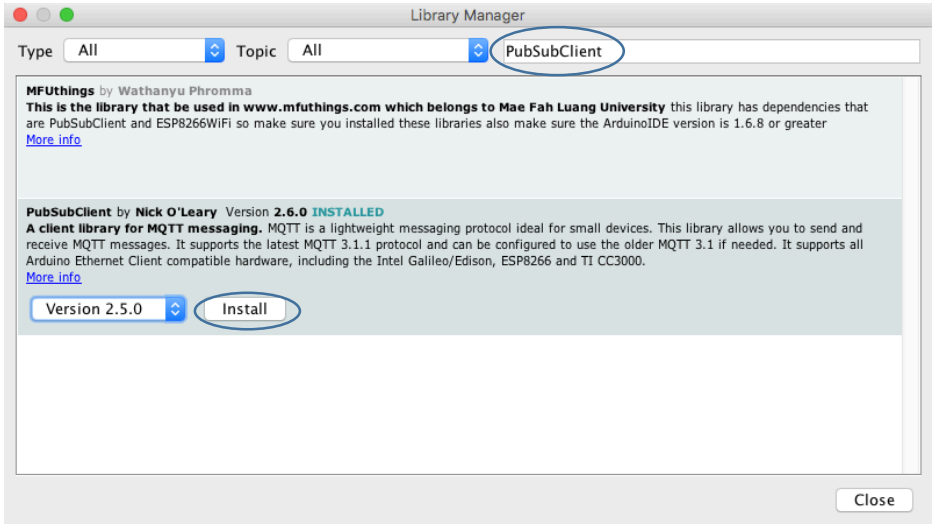



*Abbildung 115: Arduino IDE Library Manager*

Im Code für den Sketch werden nun die Bibliotheken *<PubSubClient.h>* und für das WLAN *<WiFi.h>* eingebunden. Die Credentials für die Netzwerkverbindung und die Verbindung zum Broker werden zur besseren Übersichtlichkeit in globalen Variablen angelegt:

```
#include <WiFi.h>
#include <PubSubClient.h>

const char* ssid = "SSID";
const char* password = "WIFIPASSWORD";
```

```
const char* mqttServer = "10.0.1.10";
const int mqttPort = 1883;
const char* mqttUser = "user1";
const char* mqttPassword = "userpassword1";
```

Um eine Verbindung zu dem festgelegten Server herstellen zu können, muss ein Objekt der Klasse *WiFiClient* angelegt werden. Zusätzlich wird ein Objekt der Klasse *PubSubClient* angelegt, dem der WiFiClient übergeben wird:

```
WiFiClient espClient;
PubSubClient client(espClient);
```

In der *setup()*-Funktion wird zu Beginn eine serielle Verbindung geöffnet, um unter der gewünschten Baudrate (115200) einen Output generieren zu können. Als nächstes wird die Verbindung zu dem oben definierten Netzwerk hergestellt:

```
Serial.begin(115200);
WiFi.begin(ssid, password);

while (WiFi.status() != WL_CONNECTED) {
  delay(500);
  Serial.println("Connecting to WiFi..");
}

Serial.println("Connected to the WiFi network");
```

Wenn die Verbindung zum Netzwerk erfolgreich etabliert wurde, soll die Verbindung zum MQTT Broker hergestellt werden. Dafür werden dem Object PubSubClient mit der Funktion *setServer()* die Server-Adresse und der Port mitgegeben. Mit der *connect()*-Funktion werden die Client-ID, der Benutzername und das Passwort übergeben. Wenn die Verbindung nicht erfolgreich war, wird der Grund für das Scheitern des Verbindungsaufbaus mit der *state()*-Funktion ausgegeben:

```
client.setServer(mqttServer, mqttPort);

while (!client.connected()) {
  Serial.println("Connecting to MQTT...");

  if (client.connect("ESP32Client", mqttUser, mqttPassword )) {

    Serial.println("connected");

  } else {

    Serial.print("failed with state ");
    Serial.print(client.state());
    delay(2000);

  }
}
```

Anschließend wird testweise ein Topic mit dem Topic Name `esp/test` und dem Payload "Hello from ESP32" an den Broker veröffentlicht. Dies geschieht durch die Funktion *publish()*:

```
client.publish("esp/test", "Hello from ESP32");
```

Die in einem Programm benötigte *loop()*-Hauptmethode beinhaltet die *loop()*-Funktion des PubSubClient. Diese dient dazu, regelmäßig für Nachrichten vom Broker erreichbar zu sein und mit diesem verbunden zu bleiben:

```
client.loop();
```

Der komplette Code zu diesem Test-Programm befindet sich im Anhang (client_test). Nach dem Kompilieren und dem Upload des Programms auf den ESP32 sollte auf dem Seriellen Monitor folgender Output zu abzulesen sein:

```
Connecting to WiFi..
Connected to the WiFi network
Connecting to MQTT...
connected
```

Wenn kein Fehler aufgetreten ist, kann auf dem Client, auf dem zuvor die Verbindung zum Broker mit MQTT.fx hergestellt wurde, das Topic `esp/test` abonniert werden. Nach dem Drücken auf den *Reset*-Knopf auf dem Mikrocontroller wird das Programm *client_test* auf selbigem erneut ausgeführt und in MQTT.fx wird die Nachricht "Hello from ESP32" ausgegeben:

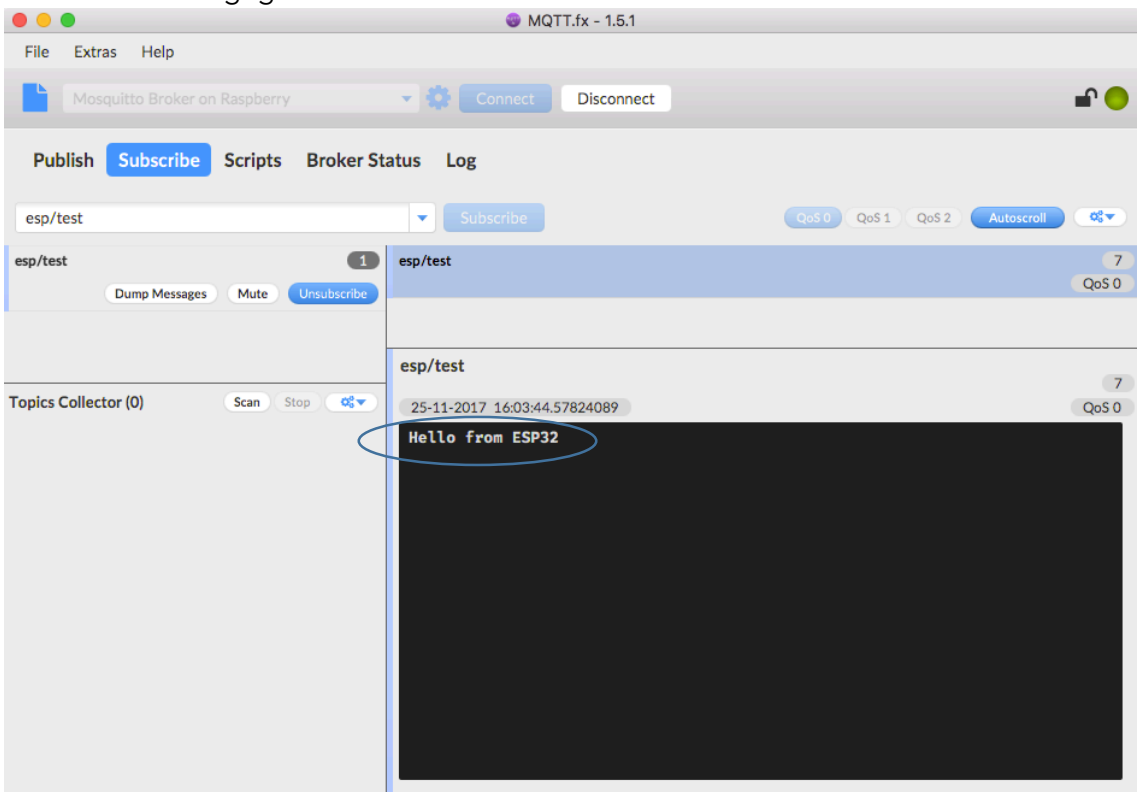



*Abbildung 116: MQTT.fx Subskription Test*

Um Werte des UV-Sensors *VEML6070* an den Broker übertragen zu können, muss dieser zunächst über die *I2C-Schnittstelle* des ESP32 angeschlossen werden. I2C (ausgesprochen: *i-squared-c*) ist ein serieller Bus, der es mehreren Geräten erlaubt, miteinander zu kommunizieren. Der I2C-Bus benötigt zwei Kommunikationskanäle, den *SDA (Serial Data)* und den *SCL (Serial Clock)*. Der ESP32 hat zwei I2C-Controller. Diese könnten im Programm variabel den GPIO Pins direkt zugeordnet werden. An dieser Stelle sollen aber die Standard-Pins verwendet werden, die vom I2C-Controller verwendet werden. Beim ESP32 lauten diese GPIO21 und GPIO22.

- Der *„Vin"-Pin* am Sensor wird mit einem Pin der Stromversorgung (*3,3V* oder *5V*) verbunden
- *G* entspricht auf dem Board Minus, also dem Pin *GND*
- *SCL* wird mit dem *GPIO Pin 22* verbunden
- *SDA* mit *Pin GPIO 21*

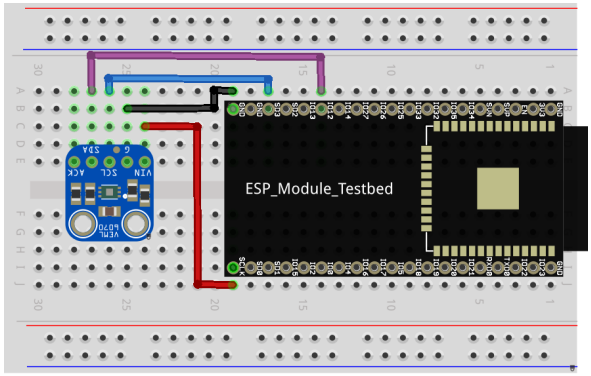



*Abbildung 117: ESP32 und VEML6070 UV-Sensor Schaltung*

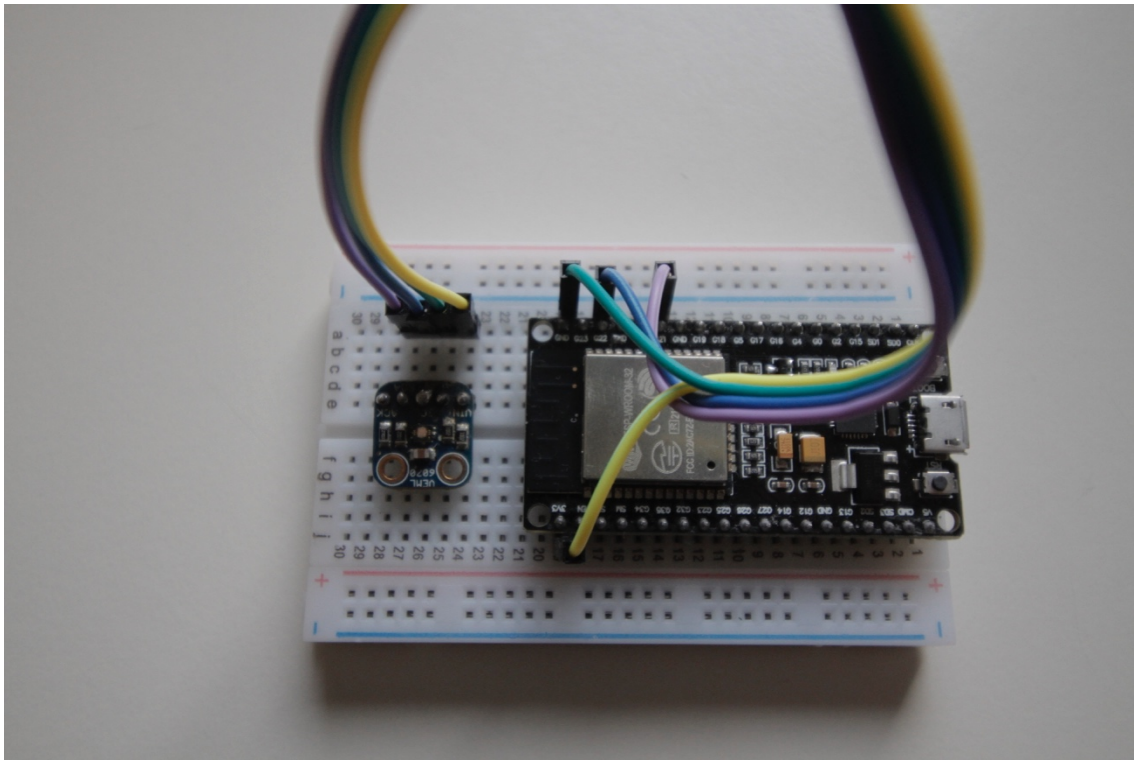


*Abbildung 118: Breadboard Verbindung von ESP32 und UV-Sensor*

Die serielle Verbindung zum Sensor soll zunächst getestet werden. Dafür wird der Arduino Sketch *i2c_scanner* kompiliert und auf den ESP32 hochgeladen (Code in Kapitel 9.5). Das Programm muss möglicherweise durch mehrmaliges Drücken des Reset-Buttons neu gestartet werden, dann sollte die Ausgabe bei der Baudrate von 115200 folgendermaßen aussehen:

```
I2C Scanner to scan for devices on each port pair D0 to D7
Scanning (SDA : SCL) - GPIO21 : GPIO22 - I2C device found at address 0x38  !
I2C device found at address 0x39  !
**********************************

Scanning (SDA : SCL) - GPIO21 : GPI32 - No I2C devices found
Scanning (SDA : SCL) - GPIO21 : GPIO33 - No I2C devices found
Scanning (SDA : SCL) - GPIO22 : GPIO21 - No I2C devices found
Scanning (SDA : SCL) - GPIO22 : GPI32 - No I2C devices found
Scanning (SDA : SCL) - GPIO22 : GPIO33 - No I2C devices found
...
```

Dies bestätigt, dass der UV-Sensor erfolgreich eingerichtet ist. Im nächsten Arbeitsschritt wird eine erste Messung durch den Sensor durchgeführt. Dafür muss die für den Sensor verfügbare *Adafruit_VEML6070*-Library unter folgendem Link heruntergeladen (github.com/adafruit/Adafruit_VEML6070/archive/master) werden und in die Arduino IDE importiert werden:

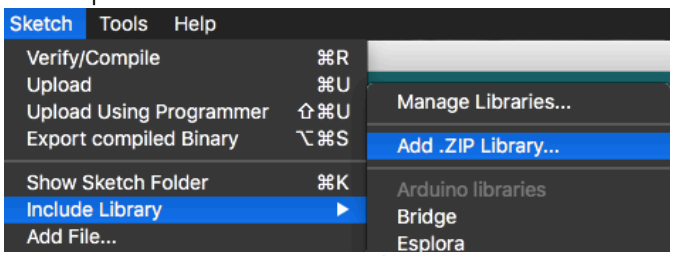



*Abbildung 119: Arduino IDE Hinzufügen einer Library*

Anschließend kann ein Beispiel-Sketch unter dem Reiter *File -> Examples (Examples from Custom Libraries) -> Adafruit_VEML6070 -> veml*test geladen werden. Nach dem Hochladen des Sketches auf den Mikrocontroller sollte nun jede Sekunde ein Wert des UV-Sensors gemessen und beim Serial Monitor ausgegeben werden. Auch hier ist es eventuell nötig, den Sketch mehrmals auf dem ESP32 zu kompilieren und gegebenenfalls zu resetten:

```
UV light level: 1
UV light level: 2
UV light level: 1
UV light level: 3
...
```

In einem geschlossenen Raum *ohne* Sonneneinstrahlung sollte der ausgegebene Wert *0* betragen. Erst wenn der Sensor UV-Strahlen erfasst, ändern sich die Werte. Für einen Test kann dieser in einen Bereich mit Tageslicht (auch bei bewölktem Himmel) gelegt werden. Dann sollte sich der Wert in einem Umfang von eins bis 5000 bewegen. Bei diesem Wert handelt es sich um die Menge an Sonneneinstrahlung im UV-Spektrum im Verhältnis Leistung pro Fläche (µW / $cm^2$).

Die zwei getesteten Komponenten MQTT Client und UV-Sensor werden nun in einem Sketch zusammengefasst. Nach dem Import der Bibliothek für den Sensor wird zu Beginn des Programms ein Objekt der Klasse *Adafruit_VEML6070* erstellt:

```
#include "Adafruit_VEML6070.h"
Adafruit_VEML6070 uv = Adafruit_VEML6070();
```

Diese Variablen definieren das verwendete Topic und legen die Länge des *char* des Payloads fest:

```
char* topic = "garden/uvsensor";
char payload[6];
String payload_str;
```

Anschließend wird in der *setup()*-Funktion des Sketches diesem Objekt eine Zeitkonstante übergeben. Diese sogenannte Integrationszeit ist für die Berechnung der Intensität verantwortlich. Je länger diese Zeit angegeben wird, desto präziser wird Licht gemessen:

```
uv.begin(VEML6070_1_T);
```

Folgende Zeitkonstanten können verwendet werden:

- VEML6070_HALF_T ~62.5ms
- VEML6070_1_T ~125ms
- VEML6070_2_T ~250ms
- VEML6070_4_T ~500ms

Wenn der Prozess des Verbindens zum WLAN in der *setup()*-Funktion abgeschlossen ist, kann in der *loop()*-Funktion folgender Code wiederholend ausgeführt werden, bei dem die Verbindung zum Broker verwaltet wird. Ebenfalls wird der ausgelesene Wert jede Sekunde in einer Variablen gespeichert und unter dem Topic Name `garden/uvsensor` an den Broker gesendet (für gesamten Code siehe Abschnitt 9.5, ab Seite 133).

```
// MQTT connection
if (!client.connected()) {
  reconnectMQTT();
}
client.loop();

//read sensor and get value as UV-Index
uint32_t uvi = getUVI(uv.readUV());
Serial.print("Sensor raw data: ");
Serial.println(uvi);

// preparing string
payload_str = String(uvi);
payload_str.toCharArray(payload, payload_str.length() + 1 );

// publish
publishToTopic(payload, topic);
```

Wie bereits zuvor schon angedeutet, misst der Sensor das UV-Licht in Mikro-Watt pro Quadrat-Zentimeter (µW/cm²). Deswegen muss der gemessene Wert noch in den offiziellen UV-Index umgerechnet werden. Dies geschieht in der *getUVI()*-Funktion, die die Umrechnung vornimmt und einen Integer-Wert zurückgibt. Der UV-Index wird normalerweise von der *International Commission on Illumination (CIE)* festgelegt und durch folgende Formel errechnet:

$$I_{UV} = k_{er} \cdot \int_{250\,nm}^{400\,nm} E_{\lambda} \cdot s_{er}(\lambda) d\lambda$$

*Abbildung 120: UV-Index Formel [69]*

Der Referenzwert des UV-Index von 0.4 entspricht einem gemessenen Wert von 0.01 W/cm². Mit diesem wird die Umrechnung vollzogen [70]:

```
int getUVI(uint32_t uv) {
  float uvi = refVal * (uv * 5.625) / 1000;
  return (int)uvi;
}
```

Um zu überprüfen, ob die Werte des UV-Sensors beim Broker durch eine Subskription zu beziehen sind, wird mithilfe von MQTT.fx das Topic garden/uvsensor abonniert. Dann sollte die Ausgabe der Werte wie folgt aussehen:

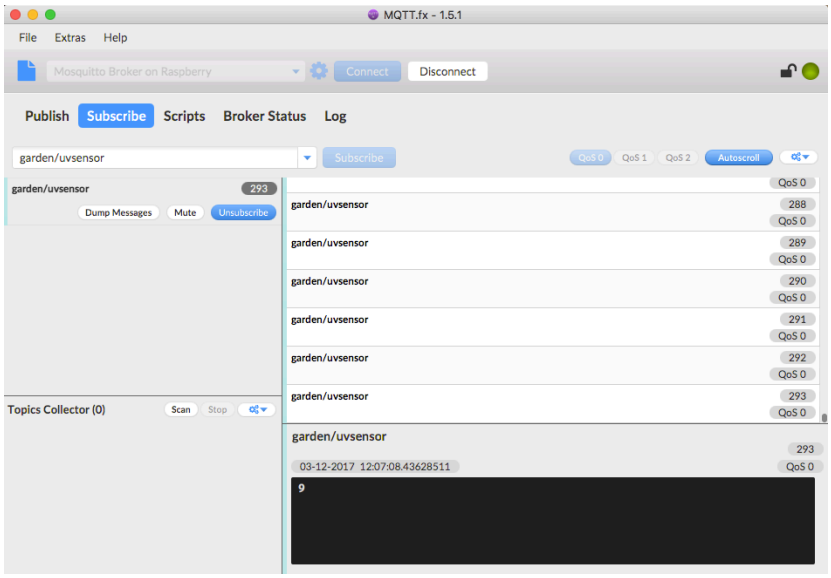



*Abbildung 121: MQTT.fx UV-Sensor Daten*

### III. Einrichten des Clients auf dem ESP32 mit NeoPixel Anzeige

Um den NeoPixel Ring mit dem ESP32 zu betreiben, wird dieser wie in folgendem Schaltplan gezeigt an den Mikrocontroller angeschlossen:

- *5V DC Power* am Ring wird mit einem Pin der Stromversorgung *(5V)* verbunden
- *GND* am Ring wird mit *GND* am Mikrocontroller verbunden
- *Data Input* wird mit dem *GPIO Pin 22* verbunden

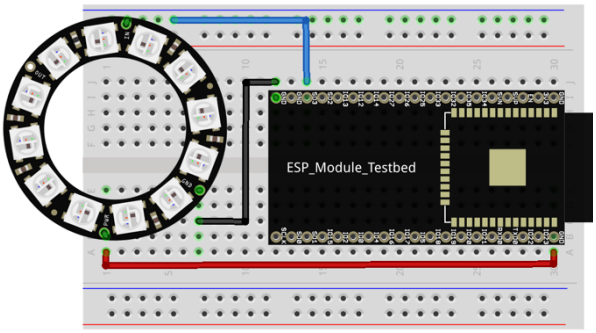



*Abbildung 122: ESP32 und NeoPixel Ring Schaltung*

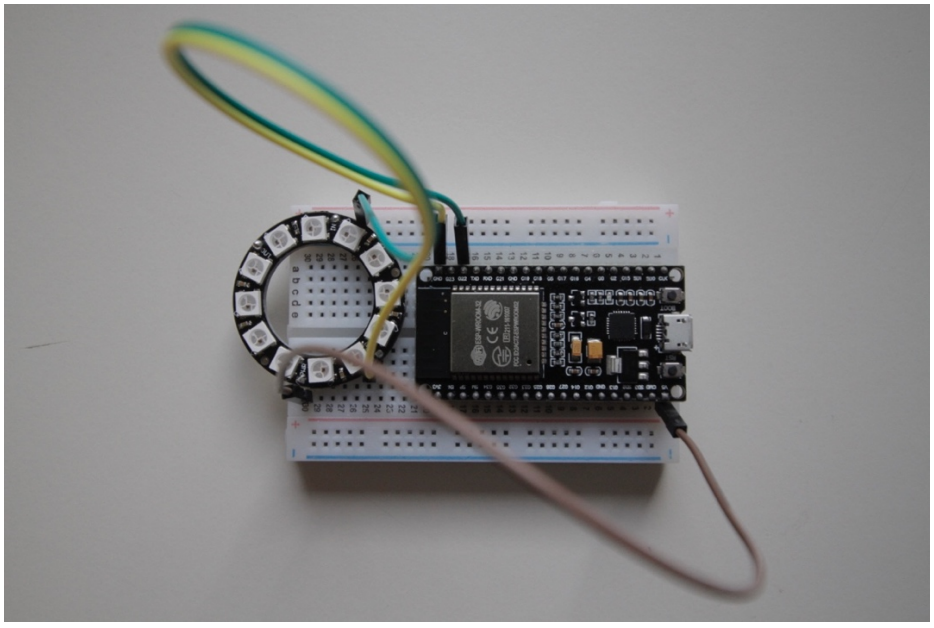


*Abbildung 123: Breadboard Verbindung von ESP32 und NeoPixel Ring*

Die Funktionalität des NeoPixel Rings wird sichergestellt, indem im ersten Schritt die *Adafruit NeoPixel Library* installiert wird:

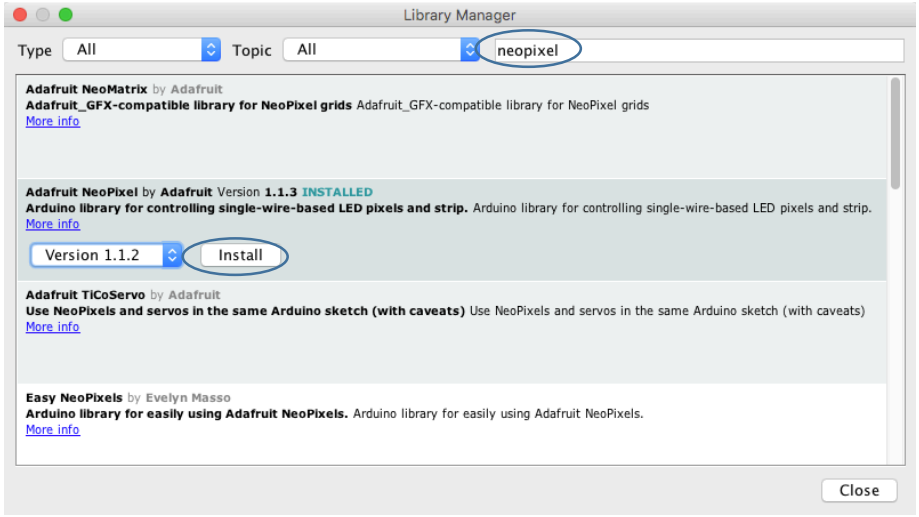



*Abbildung 124: Arduino IDE Library Manager NeoPixel Ring*

Anhand eines Test-Sketches wird geprüft, ob die NeoPixel Anzeige ordnungsgemäß funktioniert. Unter dem Menüpunkt *File -> Examples (Examples from Custom Libraries) -> Adafruit NeoPixel* findet sich das Programm *strandtest*. In diesem müssen zu Beginn mit folgenden Zeilen der GPIO Pin und die Anzahl der LEDs definiert werden:

```
#include <Adafruit_NeoPixel.h>
#define PIN 22
Adafruit_NeoPixel strip = Adafruit_NeoPixel(12, PIN, NEO_GRB +
NEO_KHZ800);
```

Nach dem Kompilieren und dem Hochladen des Sketches auf den ESP32 leuchten die LED Pixel des Rings nacheinander in unterschiedlichen Farbkombinationen auf.

Der Sketch für den Client mit NeoPixel-Anzeige, der je nach UV-Index eine andere Farbe annimmt, wird im Folgenden erstellt. Nach dem Importieren der oben schon verwendeten Libraries *<PubSubClient.h>* und *<Adafruit_NeoPixel.h>* werden einige Konstanten definiert, die den angeschlossenen GPIO Pin, die Anzahl der LEDs der Pixel-Anzeige und die Helligkeit derselben festlegen:

```
#define PIN 22
#define NUM_LEDS 12
#define BRIGHTNESS 100

Adafruit_NeoPixel strip = Adafruit_NeoPixel(NUM_LEDS, PIN, NEO_GRB +
NEO_KHZ800);

const uint32_t purple = strip.Color(134, 111, 255, 50);
const uint32_t red = strip.Color(248, 17, 22, 50);
const uint32_t orange = strip.Color(251, 116, 27, 50);
const uint32_t yellow = strip.Color(252, 199, 33, 50);
const uint32_t green = strip.Color(67, 185, 30, 50);
const int animationTime = 50;

uint32_t uvi = 1;
uint32_t uviTemp = 2;
uint32_t currentColor;
```

Der hier definierten Instanz der *Adafruit_NeoPixel*-Library werden diese Konstanten übergeben. Als nächstes werden Farben vordefiniert. Diese sollen später den fünf verschiedenen Abstufungen zugeordnet werden, die dem UV-Index entsprechen (siehe Abbildung 125, Seite 86). Eine weitere Variable ist die Zeit, die für eine Animation benötigt werden soll. Weiterhin werden der aktuelle Wert und der zuletzt gemessene Wert, sowie die aktuell verwendete Farbe in Variablen gespeichert.

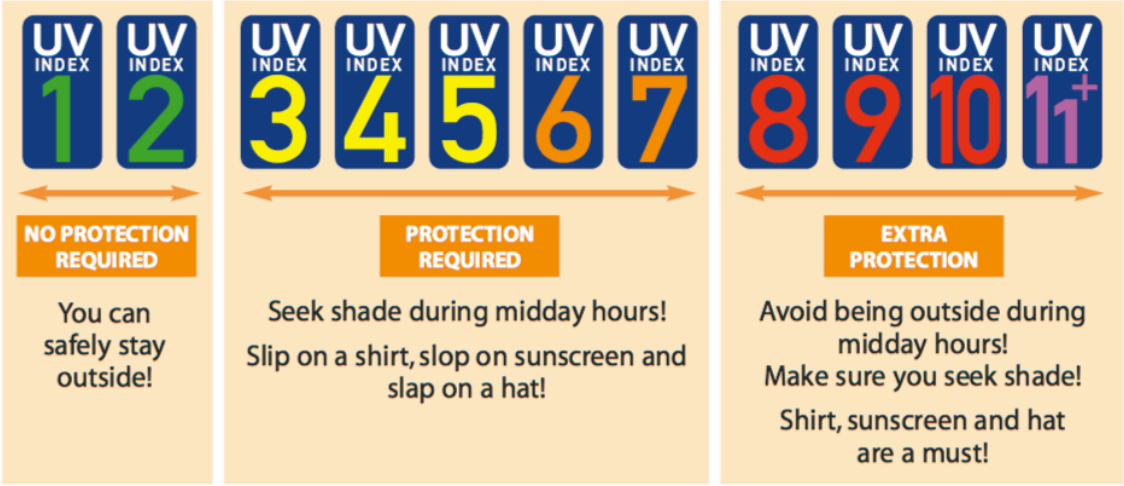



*Abbildung 125: UV-Index Indikator [69]*

Nach dem Einrichten der WiFi-Connection und dem Aufbau der Client-Server Verbindung wird in der *setup()*-Funktion des Programms für das Client-Objekt eine *callback()*-Funktion gesetzt. Weiterhin wird die zuvor definierte Helligkeit der Anzeige gesetzt und die Pixel nacheinander initialisiert, dass sie sich anfangs in ausgeschaltetem Zustand befinden:

```
Serial.begin(115200);
setup_wifi();
client.setServer(mqttServer, mqttPort);
client.setCallback(callback);

strip.setBrightness(BRIGHTNESS);
strip.begin();
strip.show();                    // Initialize all pixels to 'off'
resetColor(0);                   // set all pixels dark
```

Die *callback()*-Methode wird immer nach einer eingehenden Publish-Nachricht des Brokers aufgerufen. Sie beinhaltet die Parameter Topic, den Payload und die Länge des Payloads. Für den Payload wird ein Array des Datentyps *byte* verwendet, weswegen über selbiges iteriert und daraus ein String gebildet wird.

```
void callback(char* topic, byte* payload, unsigned int length) {
  Serial.print("Message arrived [");
  Serial.print(topic);
  Serial.println("] ");

  String message;
  for (int i = 0; i < length; i++) {
    message+=(char)payload[i];          // prepare payload message
  }

  uviTemp = message.toInt();            // convert message to integer
  Serial.print("UV-Index: ");
  Serial.println(uviTemp);
```

In der *loop()*-Methode wird der gemessene Index nun dafür verwendet, die Anzeige dementsprechend zu steuern. Dafür wird zu Beginn festgestellt, ob sich der UV-Index zu dem bisher gemessenen Wert verändert hat. Nur eine direkte Veränderung auf der Skala hat zur Folge, dass sich auch die Farbe ändert. Vor jeder Änderung der Farbe, wobei die zuvor definierten Farben verwendet werden, werden die Pixel des Rings der Reihe nach ausgeschaltet:

```
// if the current uv-index changes, change color
if (uvi != uviTemp){
  Serial.println("uv-index changed");
  uvi = uviTemp;

  // reset color
  resetColor(animationTime-20);

  if (uviTemp <= 2){
    currentColor = green;
  } else if (uviTemp <= 5){
    currentColor = yellow;
  } else if (uviTemp <= 7){
    currentColor = orange;
  } else if (uviTemp <= 10){
    currentColor = red;
  } else if (uviTemp >= 11){
    currentColor = purple;
  }
  // set current color
  colorWipe(currentColor, animationTime);
}
```

Die *colorWipe()*-Methode ist dafür verantwortlich, dass sämtliche Pixel der Anzeige nacheinander angesprochen, ihnen über die Funktion *setPixelColor()* die Farbe zugeteilt und über den Befehl *show()* die Daten an den Ring gepusht werden:

```
void colorWipe(uint32_t c, uint8_t wait) {
  for(uint16_t i=0; i<strip.numPixels(); i++) {
    strip.setPixelColor(i, c);
    strip.show();
    delay(wait);
  }
}
```

Der komplette Sketch befindet sich im Anhang unter dem Name *client_neopixel*. Nach dem Starten des Brokers und des Clients, der für das Auslesen der UV-Sensor-Daten verantwortlich ist, leuchten die Pixel des LED-Rings bei der Abwesenheit von Sonnenlicht grün, da der Sensor einen Wert von null liefert. Wenn keine starke Sonneneinstrahlung gegeben ist, lassen sich Werte auch mit MQTT.fx unter dem Topic an den Broker senden. Bei einem beispielhaften gesendeten Wert von 9 leuchtet der Ring rot, was einem UV-Index von 9 entspricht (siehe Abbildung 125, Seite 86).

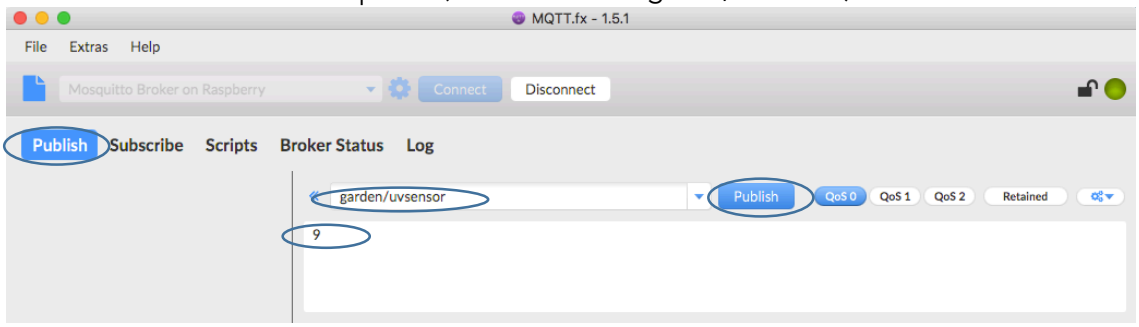



*Abbildung 126: MQTT.fx Ansprechen des NeoPixel Rings*

### IV. Erstellen der App mithilfe von Node-RED auf dem Raspberry Pi

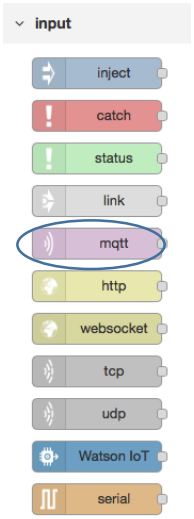
Node-RED läuft auf dem Raspberry Pi und ist dafür zuständig, das User-Interface für die App anzuzeigen. Node-RED dient dabei als Client, der das Topic subskribiert, unter dem die Werte des UV-Sensors veröffentlicht werden. Als erster Schritt bedarf es der Installation dieser visuellen Programmierumgebung über den *Node Package Manager*. Dafür werden folgende Befehle im Terminal ausgeführt. Mit dem zweiten Befehl wird bewirkt, dass Node-RED bei jedem Systemstart automatisch gestartet wird:

```
$ sudo npm install -g --unsafe-perm node-red
$ sudo systemctl enable nodered.service
```

Alternativ kann Node-RED auch über diesen Befehl gestartet werden:

```
$ node-red-start
```

Die Oberfläche der Programmierumgebung lässt sich nun im lokalen Netzwerk in einem Browser unter Eingabe der IP-Adresse und dem Port 1800 auf einem beliebigen Gerät öffnen (zum Beispiel: 10.0.1.10:1880). Programme lassen sich in Node-RED in sogenannte *Flows* unterteilen. Zunächst soll ein Flow erstellt werden, der für die Daten zuständig ist. Node-RED soll als Client dienen. Dafür muss ein sogenanntes *Node*, ein weiter unterteilter Programmteil, dafür sorgen, dass eine MQTT-Verbindung zum Broker beziehungsweise eine Subskription vorgenommen werden kann. Dafür kann aus der Palette der vorinstallierten Nodes unter dem Reiter *input* das *mqtt*-Node in den Flow gezogen werden.




Ein Node muss zunächst konfiguriert werden. Nach dem Doppelklicken auf das Node wird das Topic, das QoS Level und ein Name für das Node vergeben (siehe Abbildung 127). Im nächsten Schritt wird die Verbindung zum Broker eingerichtet. Analog wie in MQTT.fx wird der Server, der Port und unter dem Reiter *Security* der Nutzer-Namen und das Passwort eingetragen:

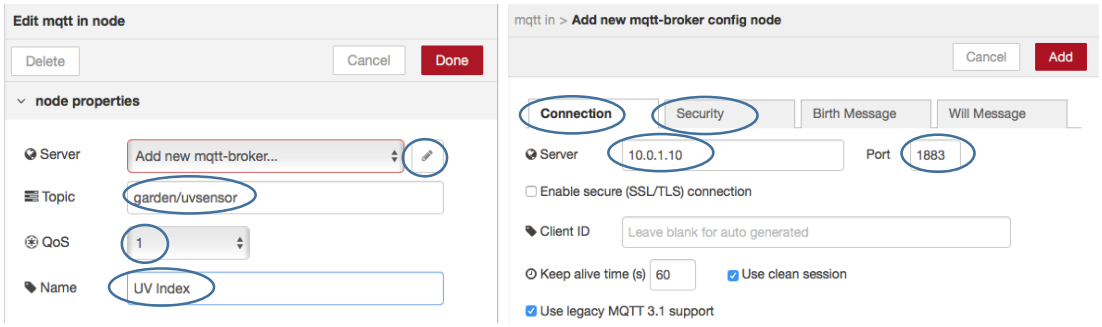



*Abbildung 127: Node-RED Node Konfiguration*

Für die Weiterverarbeitung des Wertes wird dieser in einer globalen Variable gespeichert, die über alle Flows von Node-RED verfügbar ist. Dies geschieht in der *function*-Node. Diese wird mit dem Output des MQTT-Node verbunden. In dieser wird der Payload in einen Integer konvertiert und in die globale Variable *uvIndex* geschrieben. Der Code dazu lautet:

```
msg.payload = parseInt(msg.payload);
context.global.uvIndex = msg;
return context.global.uvIndex
```

Ein *debug*-Node gibt die Werte dann an den Debugger aus. Der entworfene Flow und der Debugger sehen dann wie folgt aus, wenn Daten vom Broker gepusht werden.

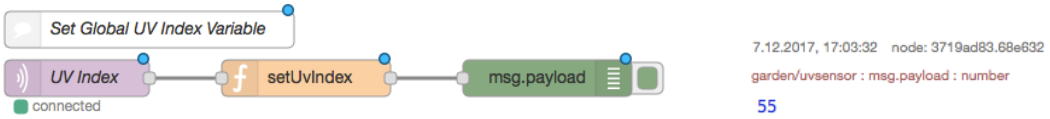
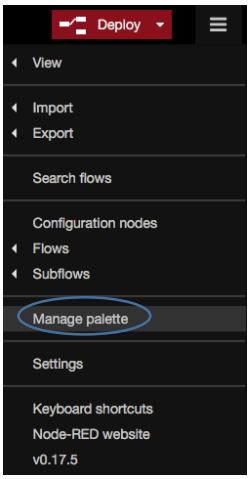



*Abbildung 128: Node-RED UV-Index Daten Flow*




Im nächsten Schritt soll von Node-RED eine UI generiert werden, die die Werte des Sensors anhand von Charts visualisiert. Dafür wird das *node-red-dashboard*-Node benötigt. Dieses wird über den *Palette Manager* installiert.

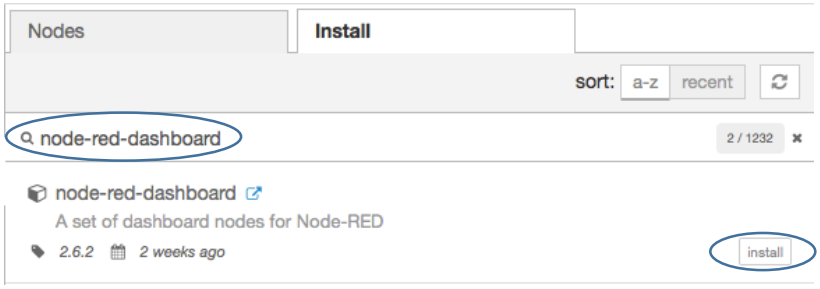
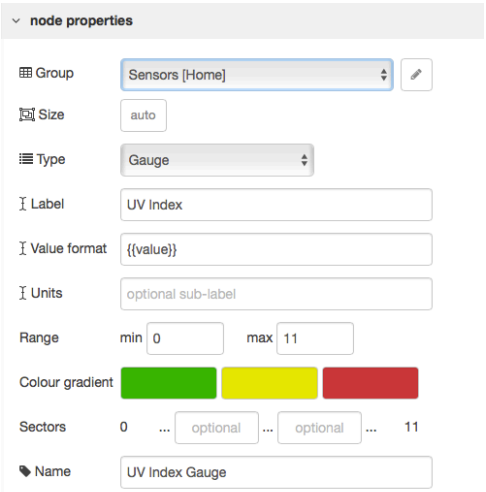



*Abbildung 129: Node-RED Palette Manager Dashboard Node*




Nach der Installation des Node wird unter *dashboard* das *gauge*-Node in einen neuen Flow gezogen. In der Konfiguration desselben wird das Anzeige-Element einer Gruppe auf der UI zugeordnet, die wiederum einem Tab untergeordnet ist. Hier werden der Name für das Label festgelegt, sowie die Einheit und der Wertebereich.

*Abbildung 130: Node-RED gauge-Node Einstellungen*

Die Anzeige muss jetzt noch in regelmäßigen Abständen mit Werten gefüllt werden. Dafür gibt die vorinstallierte *inject*-Node den richtigen Impuls, indem es in einem festgelegten Intervall einen Timestamp sendet. Die Werte werden aus der globalen Variable ausgelesen, die zuvor im anderen Flow gesetzt wurde, indem ein *function*-Node diesen Wert als *Getter* zurückliefert. Der Code dieser Funktion beschränkt sich lediglich auf diese Zeile:

```
return context.global.uvIndex
```

Der Flow sieht dann folgendermaßen aus:

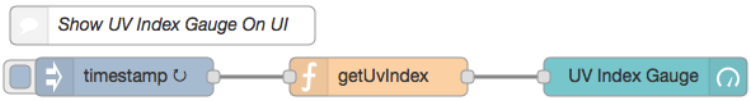



*Abbildung 131: Node-RED UV-Index UI Flow*

Nach dem Deploy-Prozess kann unter der URL von Node-RED mit dem Suffix */ui* (e.g. 10.0.1.10:1880/ui) das Dashboard aufgerufen werden, das die Anzeige enthält:


*Abbildung 132: Node-RED UI Dashboard auf dem iPhone*

## V. Anbindung zum PaaS-Dienst Adafruit IO

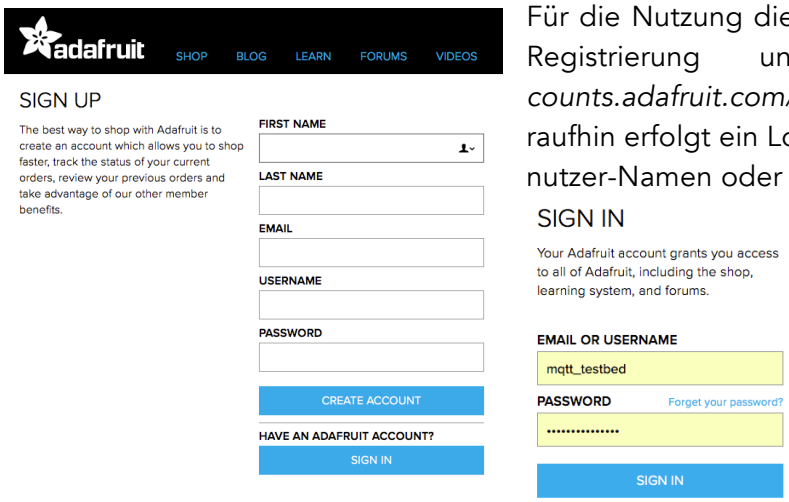



*Abbildung 133: Adafruit IO Registrieren und Einloggen*

Für die Nutzung dieses PaaS-Dienstes ist eine Registrierung unter der URL *accounts.adafruit.com/users/sign_up* nötig. Daraufhin erfolgt ein Login mit dem erstellten Benutzer-Namen oder der E-Mail-Adresse:

Unter dem Menüpunkt *Feeds* lassen sich in einer Ordnerstruktur Daten unterschiedlicher Dienste sammeln. An dieser Stelle wird ein neues Feed mit dem Namen *uvsensor* für die Subskription der Daten des UV-Sensors angelegt:

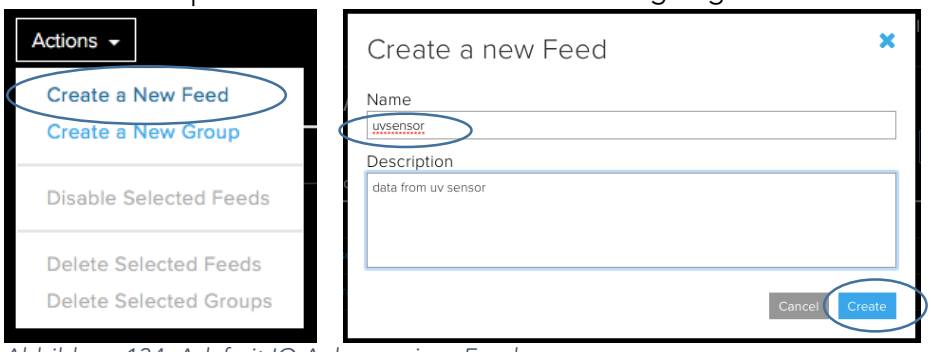
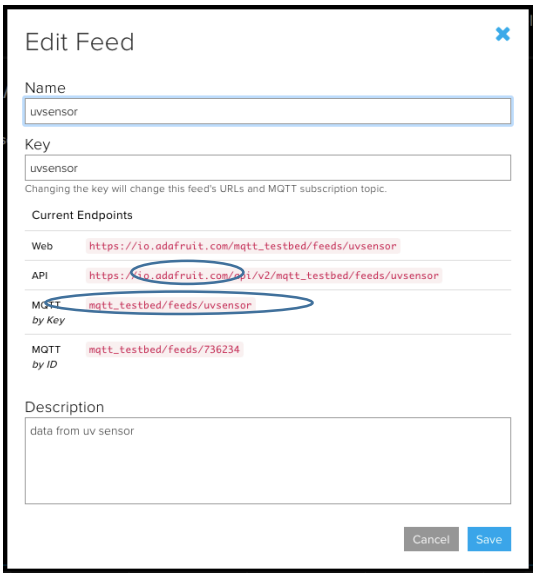



*Abbildung 134: Adafruit IO Anlegen eines Feeds*

Um zu diesem Feed Daten zu publishen, müssen die Informationen über den Server (Broker) über einen Klick auf die Schaltfläche *Feed Information* angezeigt werden. Die Adresse des Brokers lautet somit *io.adafruit.com*, der Topic Name *mqtt_testbed/feeds/uvsensor*. Für spätere Verwendung können unter dem Menüpunkt *View AIO Key* der Benutzer-Name und der *Active Key*, also das Passwort eingesehen werden.



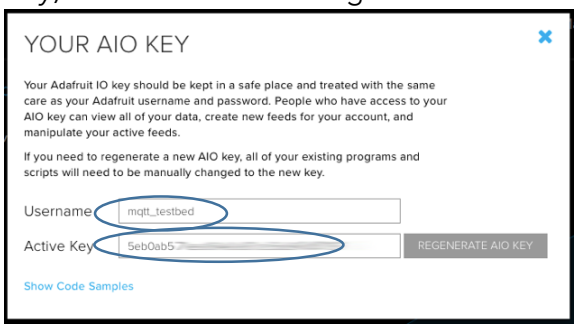




*Abbildung 135: Adafruit IO Server, Topic und Credentials*

Die Verbindung zum Broker soll mit diesen Informationen getestet werden, indem mit dem MQTT.fx Client Daten an diesen Feed gesendet werden. Für den Aufbau der Verbindung müssen die Adresse des Brokers und der Port eingegeben werden. Unter dem Reiter *Credentials* werden Username und Passwort (Active Key) eingegeben:

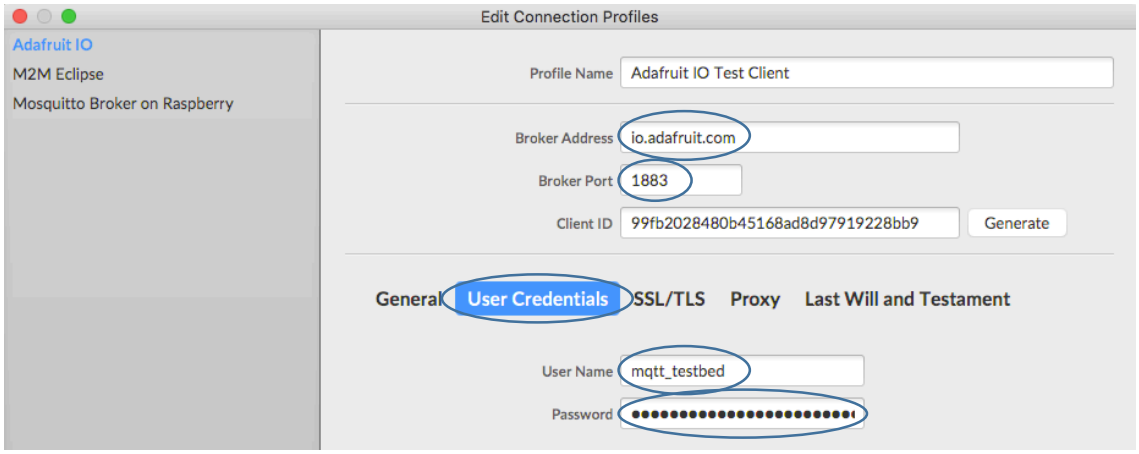



*Abbildung 136: MQTT.fx Verbindung zu Adafruit IO*

Nach dem Aufbau der Verbindung durch den Klick auf *Connect* lassen sich unter dem genannten Topic und einem beispielhaften Wert für den Payload Daten an den Adafruit Broker publishen:

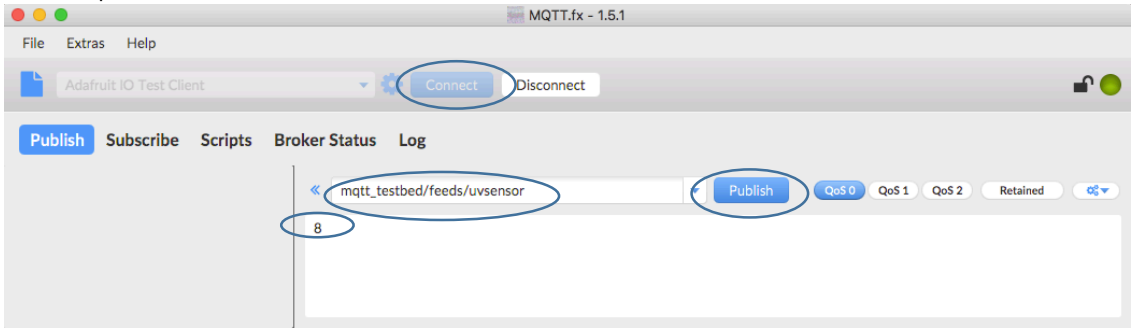



*Abbildung 137: MQTT.fx Publish zu Adafruit IO*

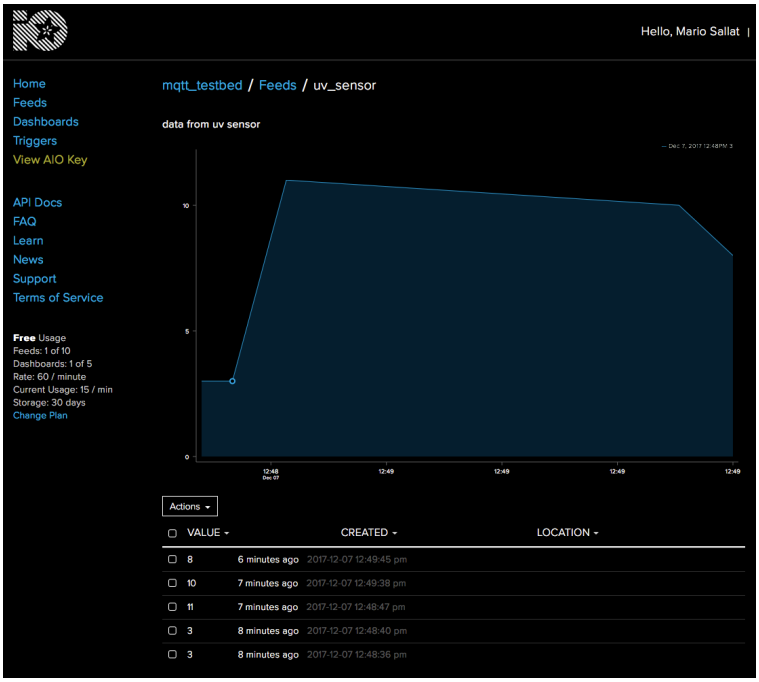
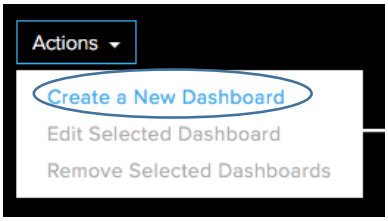



*Abbildung 138: Adafruit IO Feed Daten*

Auf der Weboberfläche von Adafruit IO sollte der Feed, der zum gleichen Topic subskribiert ist, an den der MQTT.fx Client Daten sendet, die Werte anhand eines Graphen visualisieren.




Für die Visualisierung und die Zusammenfassung aller Messdaten von verschiedenen Sensoren und Clients gibt es die Möglichkeit, diese auf einem Dashboard darzustellen. Dafür kann unter dem Menüpunkt *Dashboards* ein neues Dashboard angelegt werden.

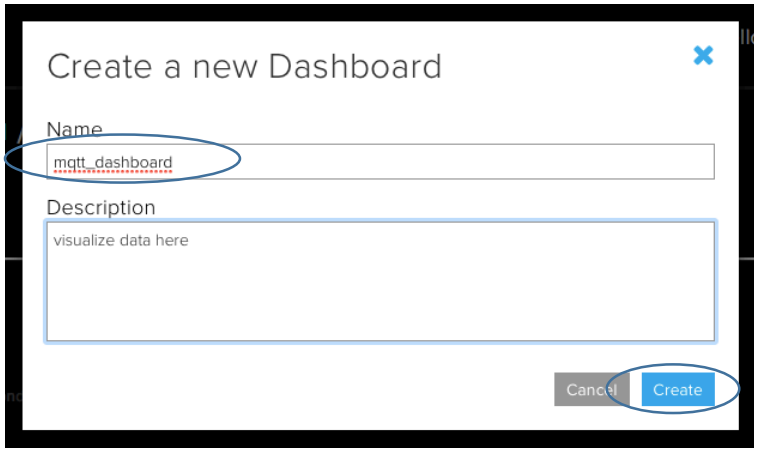



*Abbildung 139: Adafruit IO Anlegen eines neuen Dashboards*

Im nächsten Schritt wird auf dem Dashboard ein neuer sogenannter *Block* angelegt, bei dem die Art angegeben werden kann, wie die Daten visualisiert werden sollen.

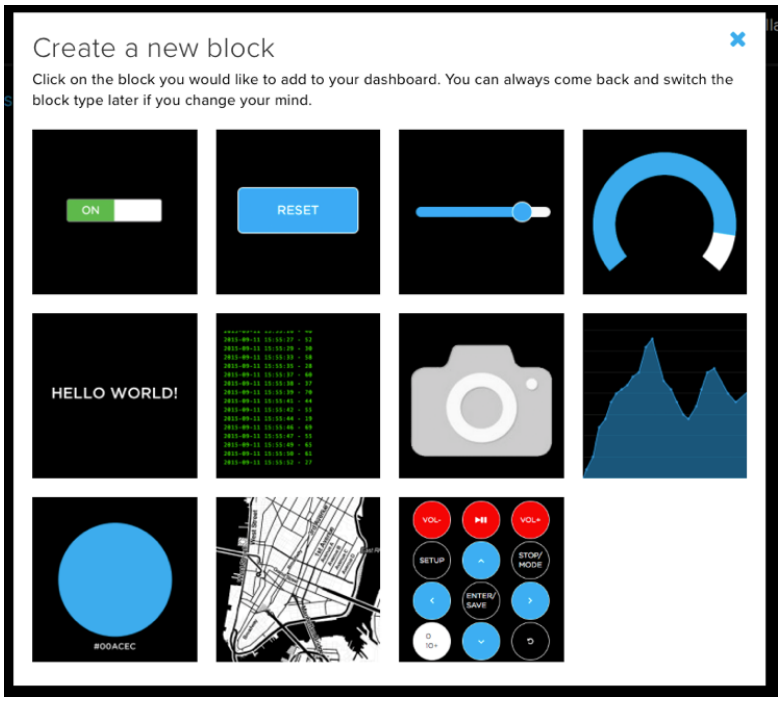




*Abbildung 140: Auswahl des Typs der Visualisierung*

Nach dem Hinzufügen der gewünschten Elemente sieht das Dashboard folgendermaßen aus. Es besteht auch die Möglichkeit, sich dieses im Vollbild-Modus anzeigen zu lassen:

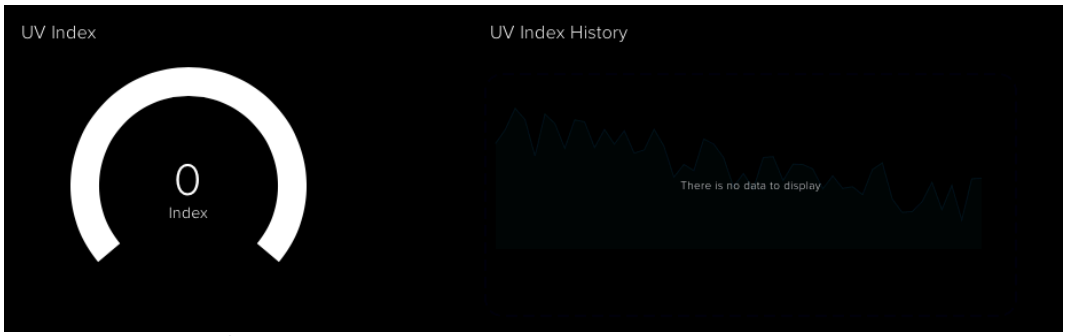



*Abbildung 141: Adafruit IO Dashboard UV-Index*

Dieser PaaS-Dienst benutzt zum Weiterleiten der Daten *nicht* den in dieser Arbeit verwendeten lokalen Mosquitto Broker. Der Broker des PaaS-Dienstes von Adafruit mit der URL *io.adafruit.com* kann durch die Möglichkeit einer *Bridge* angesprochen werden. Mit einer Bridge lassen sich über mehrere Broker hinweg Informationen vermitteln. In diesem Fall entspricht die Bridge auch einem Gateway in das Internet. Die erfassten Daten werden vom lokalen Quell-Broker an den gekoppelten Cloud-Broker gesendet. Der Ziel-Broker funktioniert prinzipiell wie ein zusätzlicher subskribierender Broker. Wenn die Informationen bei ihm eingetroffen sind, können sie von anderen Clients wie gewohnt subskribiert werden.

Um eine Kopplung der zwei Broker zu ermöglichen, muss beim Quell-Broker die Konfigurationsdatei mit folgendem Befehl geöffnet und anschließend editiert werden:

```
$ sudo nano /etc/mosquitto/conf.d/default.conf
```

In dieser werden folgende Zeilen hinzugefügt:

```
# Bridge to Adafruit.IO
connection bridge_mqtt_testbed
address io.adafruit.com:1883
try_private false
cleansession false
notifications false
remote_username USER_NAME
remote_password AIO_KEY
start_type automatic
topic uvsensor out 1 garden/ mqtt_testbed/feeds/
```

Die Bedeutungen der Einträge sind [71]:

- *connection:* Name der Bridge-Verbindung
- *adress:* URL/IP-Adresse und Port des Ziel-Brokers
- *try_private:* Signalisiert dem Ziel-Broker, dass es sich um eine Bridge handelt. Diese Option wird nicht unterstützt und muss deshalb auf *false* stehen
- *cleansession:* Clean Session bewirkt, dass ältere Daten beim Starten nicht gesetzt sind
- *notification:* verhindert das Publizieren auf geschützte Topics des Ziel-Brokers
- *remote_username:* Der Benutzername beim Adafruit Broker
- *remote_password:* Das Passwort, beziehungsweise der AIO Key, der unter *View AIO Key* eingesehen werden kann (siehe Abbildung 135, Seite 91).
- *start_type:* Der Wert *automatic* bewirkt, dass die Bridge automatisch mit dem Broker gestartet und bei Unterbrechungen neu gestartet wird
- *topic:* Mappen des Topics des lokalen Broker auf das Topic des Ziel-Brokers: topic <local topic> <in | out | both> <QoS> <local topic prefix> <remote topic prefix>

Anschließend muss der Mosquitto Broker neu gestartet werden:

```
$ sudo /etc/init.d/mosquitto stop
$ sudo /etc/init.d/mosquitto start
```

Im Dashboard auf *io.adafruit.com/mqtt_testbed/dashboards* werden die Daten nun visualisiert:

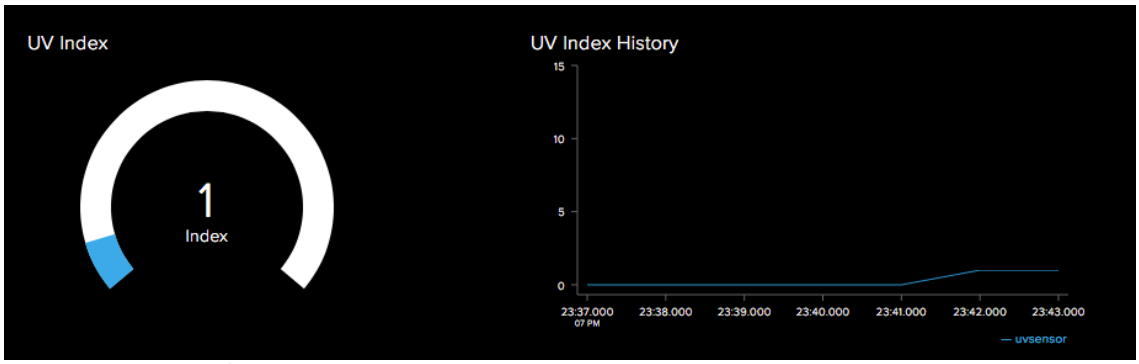



*Abbildung 142: Adafruit IO Dashboard Visualisierung der Daten (UV-Index)*

### 4.4.2. Anwendung 2: Luftqualität Messstation

**VI. Einrichten des Clients auf dem ESP32 mit Luftqualität-Sensor**

Der Adafruit CSS811 VOC Sensor für die Messung der Luftqualität wird wie der UV-Sensor der vorigen Anwendung über die serielle I2C-Schnittstelle angeschlossen. Dafür werden folgende Anschlüsse miteinander verbunden:

- *Vin* ist der Pin für die Stromversorgung. An diesen kann *3,3* oder *5 Volt* angeschlossen werden, da ein internes Bauteil die Spannung auf 3,3 Volt reguliert
- *GND* wird an *GND* an den ESP32 angeschlossen
- *SDA* ist der I2C Clock Pin, der mit dem *GPIO 21* verbunden wird
- *SCL* muss mit dem *GPIO 22* Pin des Mikrocontrollers angeschlossen werden
- Der *Wake* Pin wird ebenfalls mit *Ground* verbunden

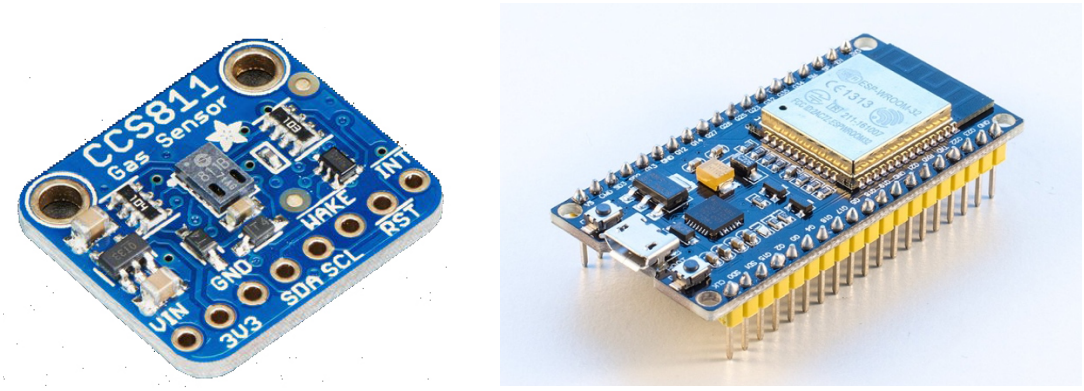



*Abbildung 143: CSS811 Gas Sensor und ESP32 Mikrocontroller*

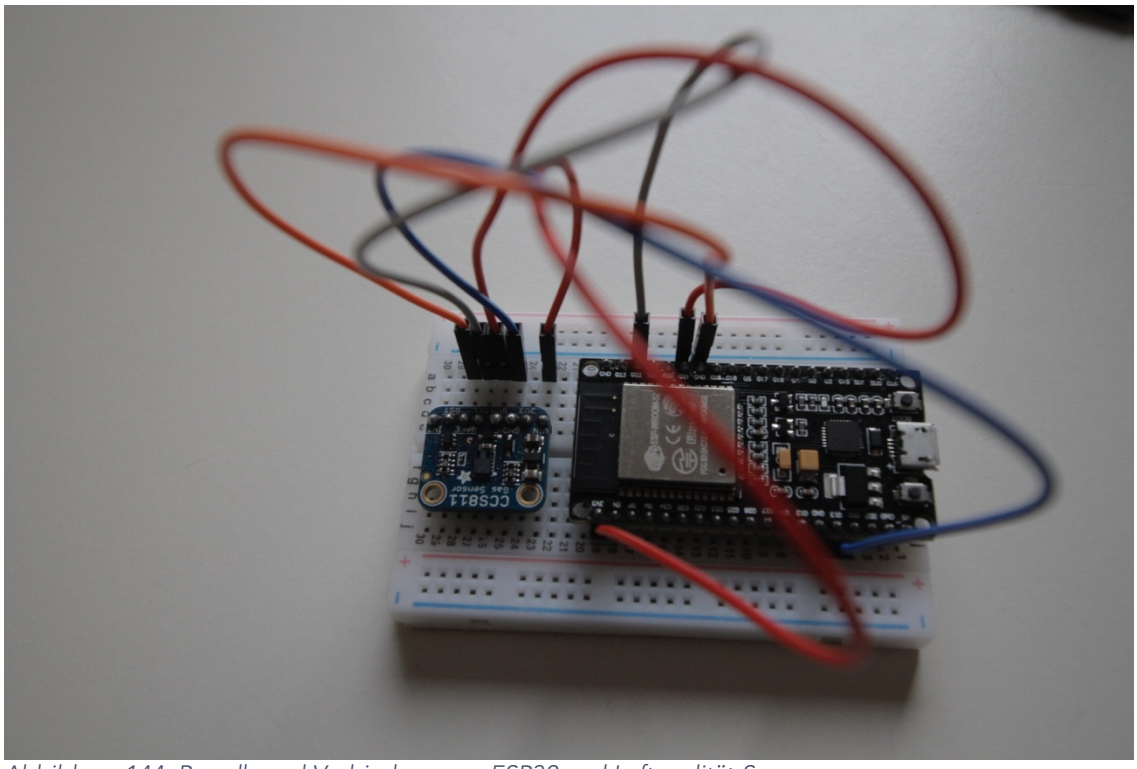


*Abbildung 144: Breadboard Verbindung von ESP32 und Luftqualität-Sensor*

Um die erste Messung mit dem Sensor durchführen zu können, muss die *Adafruit_CSS811*-Library heruntergeladen und eingebunden werden. Diese ist unter folgendem Link auf Github verfügbar: *github.com/adafruit/Adafruit_CCS811/archive/master.zip*. Nach dem Importieren der Bibliothek in die IDE (siehe Abbildung 119, Seite 81) kann unter dem Reiter *File -> Examples (Examples from Custom Libraries) -> Adafruit CSS811 Library* der Sketch *CSS811_test* gefunden werden. Nach dem Kompilieren und Hochladen auf den Mikrocontroller, dem Öffnen des seriellen Monitors und dem Einstellen der Baudrate auf 9600 sieht die Ausgabe etwa wie folgt aus:

```
CSS811 test
CO2: 0ppm, TVOC: 0ppb   Temp:25.00
CO2: 0ppm, TVOC: 0ppb   Temp:25.00
CO2: 0ppm, TVOC: 0ppb   Temp:25.00
CO2: 400ppm, TVOC: 0ppb   Temp:25.00
CO2: 400ppm, TVOC: 0ppb   Temp:25.00
CO2: 409ppm, TVOC: 1ppb   Temp:25.00
```

Der Sensor ist somit fertig eingerichtet und es bedarf eines Sketches, der den ESP32 als Client die Werte an den Broker publizieren lässt. In diesem Programm wird zu Beginn die zuvor installierte Bibliothek importiert und ein Objekt derselben erstellt:

```
#include "Adafruit_CCS811.h"
Adafruit_CCS811 ccs;
```

In der *setup()*-Funktion des Sketches wird der Sensor mit der Methode *begin()* gestartet und der Wert für die Temperaturmessung initialisiert:

```
if(!ccs.begin()){
  Serial.println("Failed to start sensor! Check your wiring.");
  while(1);
}

//calibrate temperature sensor
while(!ccs.available());
float temp = ccs.calculateTemperature();
ccs.setTempOffset(temp - 25.0);
```

In der *loop()*-Methode werden im Intervall von einer Sekunde die Werte des Sensors ausgelesen und an den Server gepublished. Wenn der CSS811 Sensor verfügbar ist und der Sensor momentan keine Werte ausliest, wird zuerst der ausgelesene Wert für das $CO_2$-Äquivalent in der Variablen *eCO2* gespeichert, und anschließend als char-Array an den Broker unter dem angegebenen Topic veröffentlicht:

```
// read sensor data
if(ccs.available()){
  if(!ccs.readData()){

    // read CO2
    int eCO2 = ccs.geteCO2();
    //prepare char
    payload_str = String(eCO2);
    payload_str.toCharArray(payload, payload_str.length() + 1 );
    // publish CO2
    publishToTopic(payload, "room/airquality/co2");
```

Darauf folgt der Wert, der die Summe der flüchtigen organischen Verbindungen der Umgebungsluft darstellt *(TVOC)*:

```
    // read TVOC
    int tvOC = ccs.getTVOC();
    // prepare char
    payload_str = String(tvOC);
    payload_str.toCharArray(payload, payload_str.length() + 1 );
    // publish TVOC
    publishToTopic(payload, "room/airquality/tvoc");
```

Auch die Umgebungstemperatur wird nach dem Auslesen an den Broker gepublished:

```
    // read temperature
    float temperature = ccs.calculateTemperature();
    // prepare char
    payload_str = String(temperature);
    payload_str.toCharArray(payload, payload_str.length() + 1 );
    // publish temperature
    publishToTopic(payload, "room/airquality/temperature");
```

Die Ausgabe auf dem seriellen Monitor zeigt nun etwa folgenden Output:

```
Connected to the WiFi network
Connecting to MQTT...
connected
Sending payload: 817 to topic: room/airquality/co2
Publish ok
Sending payload: 65 to topic: room/airquality/tvoc
Publish ok
Sending payload: 25.59 to topic: room/airquality/temperature
Publish ok
```

### VII. Einrichten des Clients auf dem ESP32 mit Motor für Fenstersteuerung

Der Servomotor soll bei dem Überschreiten bestimmter Messwerte der Umgebungsluft seine Stellung verändern. Zuerst werden die Anschlüsse des Motors mit den Pins des Mikrocontrollers verbunden:

- *Plus (rot)* mit *3,3 Volt*
- *Minus (schwarz)* mit *GND*
- *Analog In* mit *GPIO PIN 25*

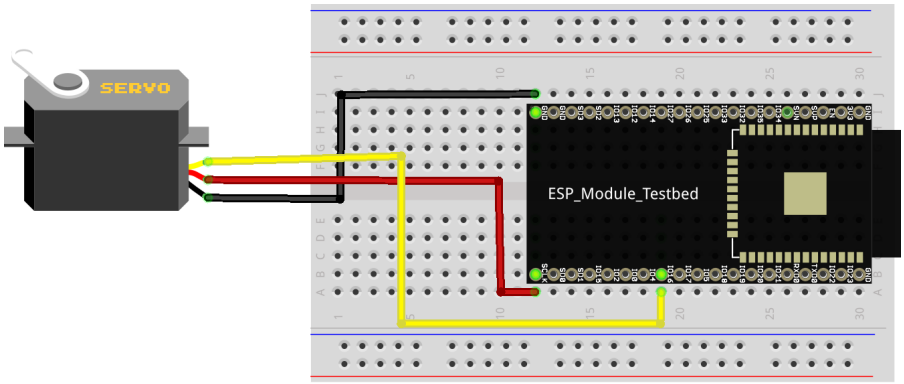



*Abbildung 145: ESP32 und Servomotor Schaltung*

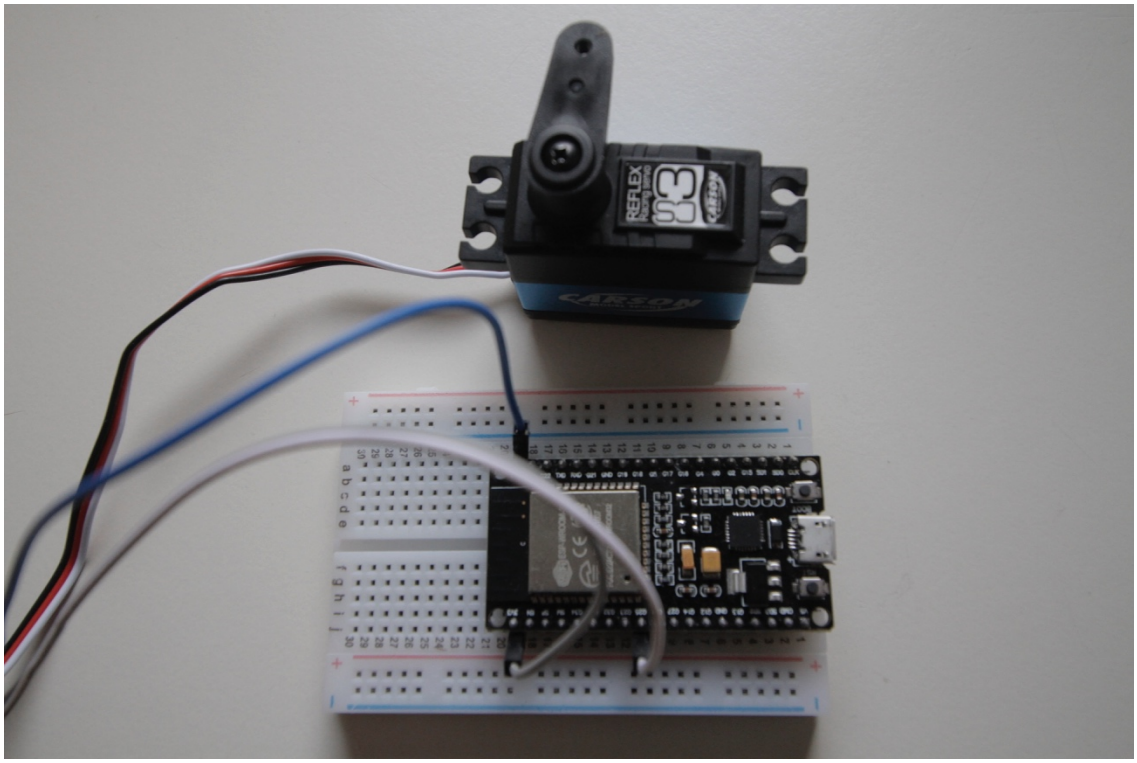


*Abbildung 146: Verbindung von ESP32 und Servomotor*

Die Funktionalität des Motors kann nach dem Importieren der *ESP32_Servo*-Library (siehe Abbildung 147) für den ESP32 getestet werden:

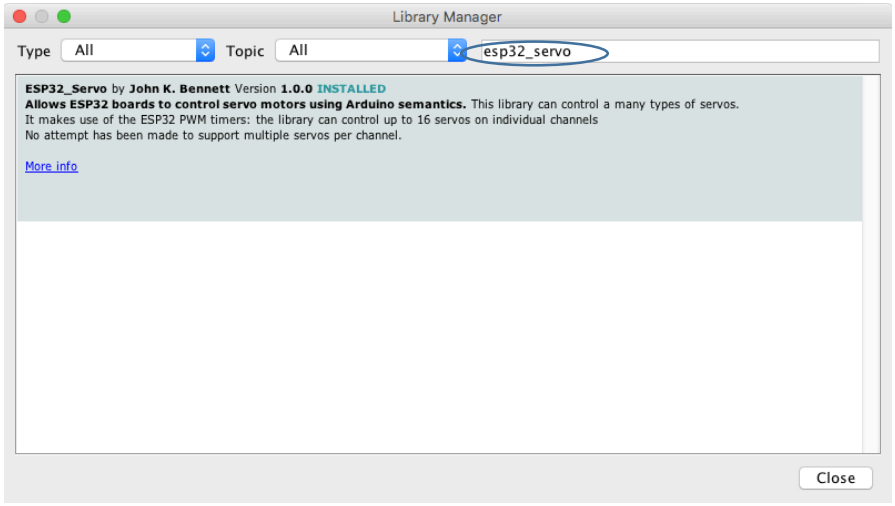



*Abbildung 147: Arduino IDE Import der Servo Library*

Dafür kann unter der Schaltfläche *File -> Examples (Examples from Custom Libraries) -> ESP32_Servo* der Sketch *Sweep* auf dem ESP32 ausgeführt werden. Die *Variable servoPin* wird auf den Wert 25 gesetzt, da über diesen Pin der Motor gesteuert wird. Nach dem Hochladen auf den Mikrocontroller dreht sich der Motor von einer Stellung zur anderen und zurück.

Der Sketch, der dafür verantwortlich ist, dass dieser Client die Topics `room/airquality/tvoc` und `room/airquality/eco2` subskribiert, muss wie gehabt die *PubSubClient*-Library importieren. Ein weiterer Import ist die Bibliothek für den Servomotor:

```
#include <PubSubClient.h>
#include <ESP32_Servo.h>
```

Das Programm verwendet für die Logik und das Speichern der Daten die folgenden Variablen. Die Variable *measurementPeriod* legt die Länge des Intervalls fest, in dem die Sensorwerte in einem Array gespeichert werden. Die Daten werden jede Sekunde vom Broker gepublished, also wird bei *measurementPeriod = 60* über einen Zeitraum von einer Minute gemessen und dann der Durchschnittswert errechnet. Wenn dieser Wert dann über dem festgelegten Schwellenwert *(eCO2Threshold, tvOCShreshold)* liegt, wird das Fenster geöffnet, beziehungsweise der Servomotor in Bewegung gesetzt. Wenn der Durchschnittswert unter den Schwellenwert sinkt, schließt sich das Fenster.

```
const int measurementPeriod = 60;

char* tvOCtopic = "room/airquality/tvoc";
int eCO2Values[measurementPeriod];
int eC02PeriodIndex = 0;
const int eCO2Threshold = 600;

char* eCO2Topic = "room/airquality/eco2";
int tvOCValues[measurementPeriod];
int tvOCPeriodIndex = 0;
const int tvOCThreshold = 25;
```

Der aktuelle Status, der beschreibt, ob das Fenster geöffnet oder geschlossen ist, wird mit der Variable *isWindowOpen* beschrieben. Von der zuvor importierten Bibliothek für

den Motor wird nun ein Objekt erstellt und der Pin definiert, über den der Motor gesteuert wird:

```
int isWindowOpen = false;

Servo myservo;
int servoPin = 25;
```

In der *setup()*-Methode wird neben dem Aufbau der WiFi-Connection und dem Übergeben von Variablen an den MQTT Client dem Servomotor-Objekt der oben definierte Pin zugewiesen:

```
void setup() {
  Serial.begin(115200);
  myservo.attach(servoPin);
  setup_wifi();
  client.setServer(mqttServer, mqttPort);
  client.setCallback(callback);
}
```

Die *callback()*-Funktion wird bei jeder empfangenen Nachricht zu jedem subskribierten Topic aufgerufen:

```
void callback(char* topic, byte* payload, unsigned int length) {
  String message;
  for (int i = 0; i < length; i++) {
    // prepare payload message
    message+=(char)payload[i];
  }
```

An dieser Stelle wird überprüft, ob das aktuelle Topic sich auf die TVOC-Werte bezieht. Wenn sie diesem Topic zugeordnet sind, wird das zugehörige Array mit Daten befüllt. Die gleiche Überprüfung geschieht jedes Mal auch bezogen auf den $CO_2$-Äquivalent-Wert:

```
  // tvOC handling
  if(strcmp(topic, tvOCtopic) == 0){
    tvOCValues[tvOCPeriodIndex] = message.toInt();
    Serial.print("Value: ");
    Serial.println(tvOCValues[tvOCPeriodIndex]);
    tvOCPeriodIndex++;
  }

  // eCO2 handling
  else if(strcmp(topic, eCO2Topic) == 0){
    eCO2Values[eC02PeriodIndex] = message.toInt();
    Serial.print("Value: ");
    Serial.println(eCO2Values[eC02PeriodIndex]);
    eC02PeriodIndex++;
  }
}
```

In der *loop()*-Methode wird zuerst global überprüft, ob die Periode, in der die Messungen durchgeführt wurden, abgelaufen ist. Wenn als nächstes das Fenster den Status *geschlossen* hat, der durchschnittliche $CO_2$-Äquivalent-Wert und der TVOC-Wert im Durchschnitt über den jeweiligen Schwellenwerten liegen, wird der Motor in Bewegung gesetzt. Wenn der Schwellenwert einer Periode wieder unterschritten wird, wird das Fenster wieder geschlossen.

```
if (eC02PeriodIndex == measurementPeriod){
    if (calculateAverage(eCO2Values) > eCO2Threshold &&
        calculateAverage(tvOCValues) > tvOCThreshold &&
        isWindowOpen == false){
        openWindow(true);
    } else if (calculateAverage(eCO2Values) < eCO2Threshold &&
               calculateAverage(tvOCValues) < tvOCThreshold &&
               isWindowOpen == true){
        openWindow(false);
    }
```

Nach dem Ablaufen eines Mess-Intervalls werden die Daten-Arrays zurückgesetzt:

```
    eC02PeriodIndex = 0;
    tvOCPeriodIndex = 0;
    memset(eCO2Values, 0, sizeof eCO2Values);
    memset(tvOCValues, 0, sizeof tvOCValues);
}
```

Die Funktion *openWindow()* kümmert sich um das Schließen und Öffnen des Fensters, beziehungsweise darum, dass sich der Servomotor je nach Übergabe-Status in die ein oder andere Richtung dreht. Eine Schleife lässt den Motor Grad für Grad auf die gewünschte Position drehen:

```
void openWindow(boolean stat){
  if (stat == true){
    Serial.println("*********** open window ***********");
    isWindowOpen = true;
    for (int pos = 0; pos <= 180; pos += 1) {
      myservo.write(pos);
      delay(15);                       }
  } else {
    Serial.println("*********** close window ***********");
    isWindowOpen = false;
    for (int pos = 180; pos >= 0; pos -= 1) {
      myservo.write(pos);
      delay(15);
    }
  }
}
```

### VIII. Erweiterung der App um die Anzeige der Luftqualität

Die drei Parameter $CO_2$-Äquivalent, TVOC und Temperatur sollen auf dem Dashboard der App angezeigt werden. In der Programmierumgebung Node-RED werden die Daten dazu in dem Flow *data* in globalen Variablen gespeichert. Analog zur vorigen Vorgehensweise bei den Daten zu Anwendung 1 wird ein MQTT-Node in den Flow gezogen und konfiguriert, das sich beim Broker als Client subskribiert.

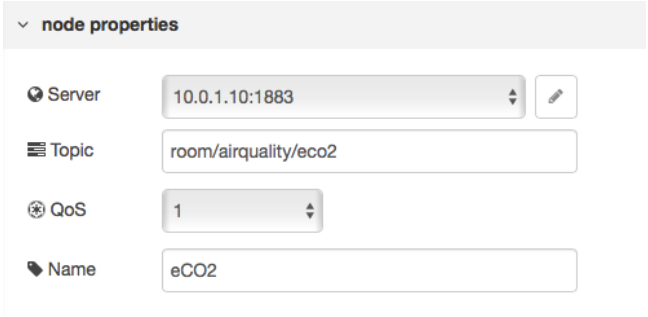



*Abbildung 148: Node-RED Konfiguration der MQTT-Node für das $CO_2$-Äquivalent*

Das MQTT-Node wird mit einer *function*-Node verbunden. In dieser wird der Payload der Nachricht in die globale Variable *eCO2* geschrieben:

```
msg.payload = parseInt(msg.payload);
context.global.eCO2 = msg;
return context.global.eCO2;
```

Eine nun verbundene *debug*-Node sorgt dafür, dass die Debug-Konsole folgendes ausgibt:


*Abbildung 149: Node-RED Debug Ausgabe des $CO_2$-Äquivalent-Wertes*

Analog dazu werden jeweils Clients für Temperatur und den TVOC-Wert erstellt und die Daten in globale Variablen gespeichert. Der Flow sieht dann dementsprechend aus:

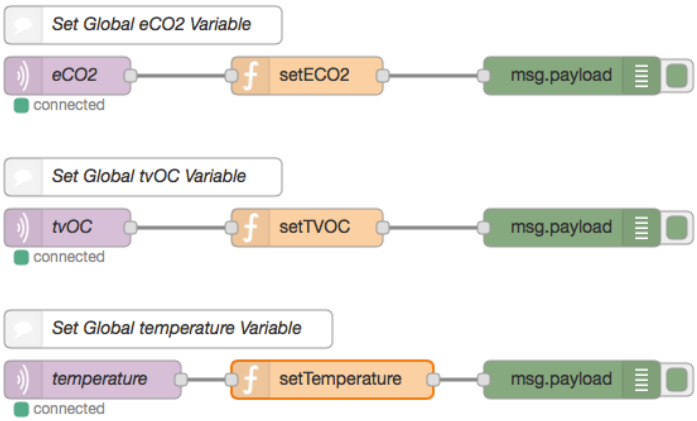



*Abbildung 150: Node-RED data Flow*

Um die UI-Elemente für das Dashboard zu generieren, wird im *ui*-Flow für die drei unterschiedlichen Werte jeweils ein eigenes Anzeige-Element (*gauge*-Node) in den Flow gezogen. Damit die Anzeige jede Sekunde mit dem aktuellen Wert aktualisiert wird, wird nach einem *inject*-Node, welches den Impuls mit einem Timestamp gibt, ein Getter eingebaut, der den aktuell gemessenen Wert der jeweiligen globalen Variable liefert. Die Getter beinhalten diesen Code:

```
return context.global.eCO2
```

beziehungsweise:

```
return context.global.tvOC
```

respektive:

```
return context.global.temperature
```

Die implementierten Nodes sehen dann wie folgt aus:

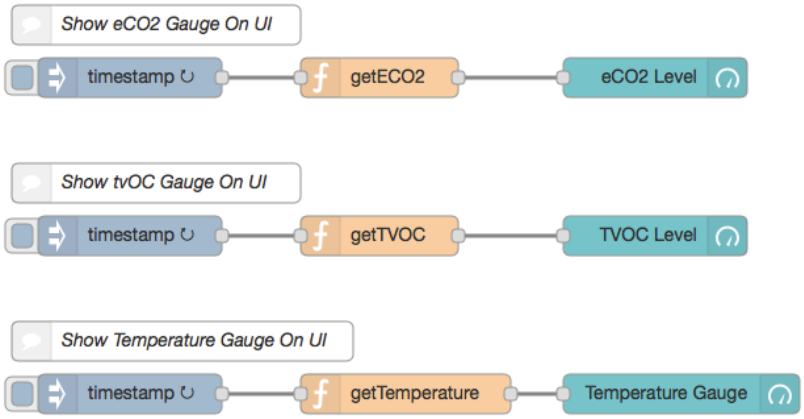



*Abbildung 151: Node-RED ui Flow*

Das fertig implementierte Dashboard sieht nach dem Ausführen des Deployments auf dem iPad wie folgt aus. Das Design ist responsive, die gruppierten Elemente würden sich also auf dem Smartphone untereinander anordnen:


*Abbildung 152: Node-RED Dashboard auf dem iPad*

## IX. Anbindung zum PaaS-Dienst Adafruit IO

Die Registrierung und die Einrichtung für die Nutzung dieses PaaS-Dienstes wurde bereits in Kapitel 4.4.1 beschrieben. Für das Empfangen der Daten der Luftqualität werden für jedes Topic unterschiedliche Feeds angelegt:

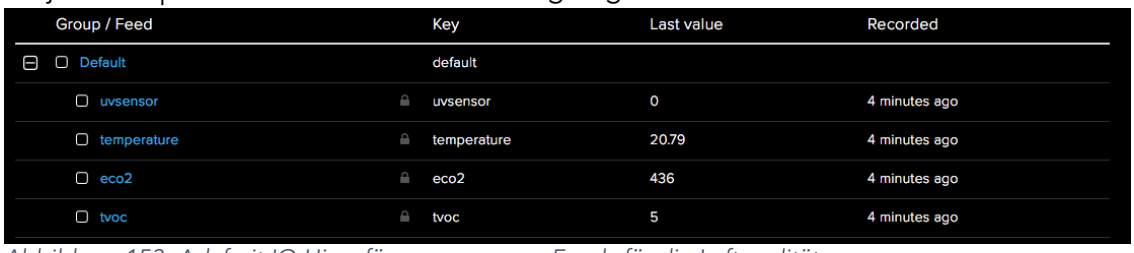



*Abbildung 153: Adafruit IO Hinzufügen von neuen Feeds für die Luftqualität*

In den *Feed-Informationen* ist das Topic einsehbar, unter dem der publizierende ESP32-Client die Daten des Luftqualität-Sensors (in diesem Fall das $CO_2$-Äquivalent) veröffentlichen muss:

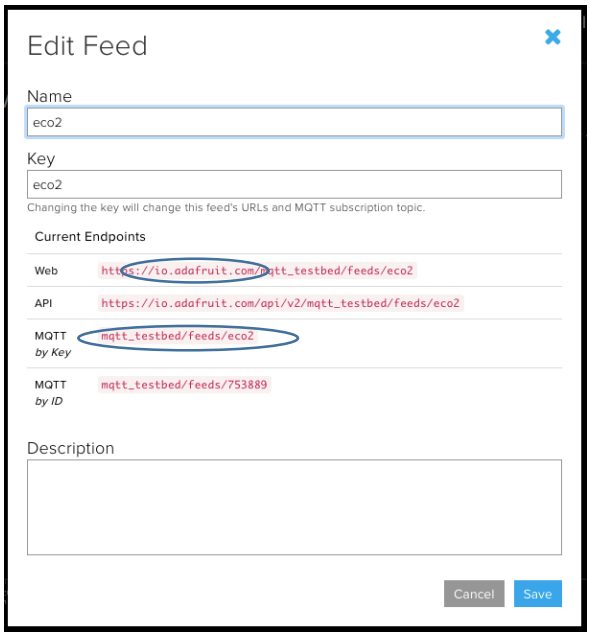



*Abbildung 154: Adafruit IO Feed Informationen*

Die anfallenden Daten der drei erstellten Feeds lassen sich auf dem Dashboard visualisieren, indem neue Blöcke unter Bezugnahme auf den betreffenden Feed hinzugefügt werden:

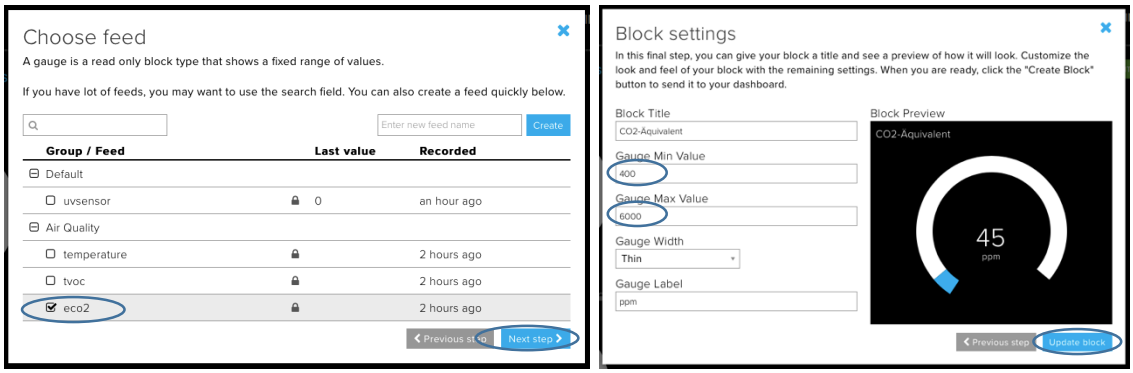



*Abbildung 155: Adafruit IO Hinzufügen eines Dashboard-Elements*

Die Konfigurationsdatei des Quell-Brokers wird beim nächsten Schritt wieder mit folgendem Befehl geöffnet:

```
$ sudo nano /etc/mosquitto/conf.d/default.conf
```

Hier werden unten in der Datei diese Zeilen hinzugefügt, um das Topic vom Quell-Broker auf das Topic des Ziel-Brokers zu mappen:

```
# Bridge to Adafruit.IO
topic eco2 out 1 room/airquality/ mqtt_testbed/feeds/
topic temperature out 1 room/airquality/ mqtt_testbed/feeds/
topic tvoc out 1 room/airquality/ mqtt_testbed/feeds/
```

Das Dashboard sieht nach dem Hinzufügen der Elemente und deren Konfiguration wie folgt aus:

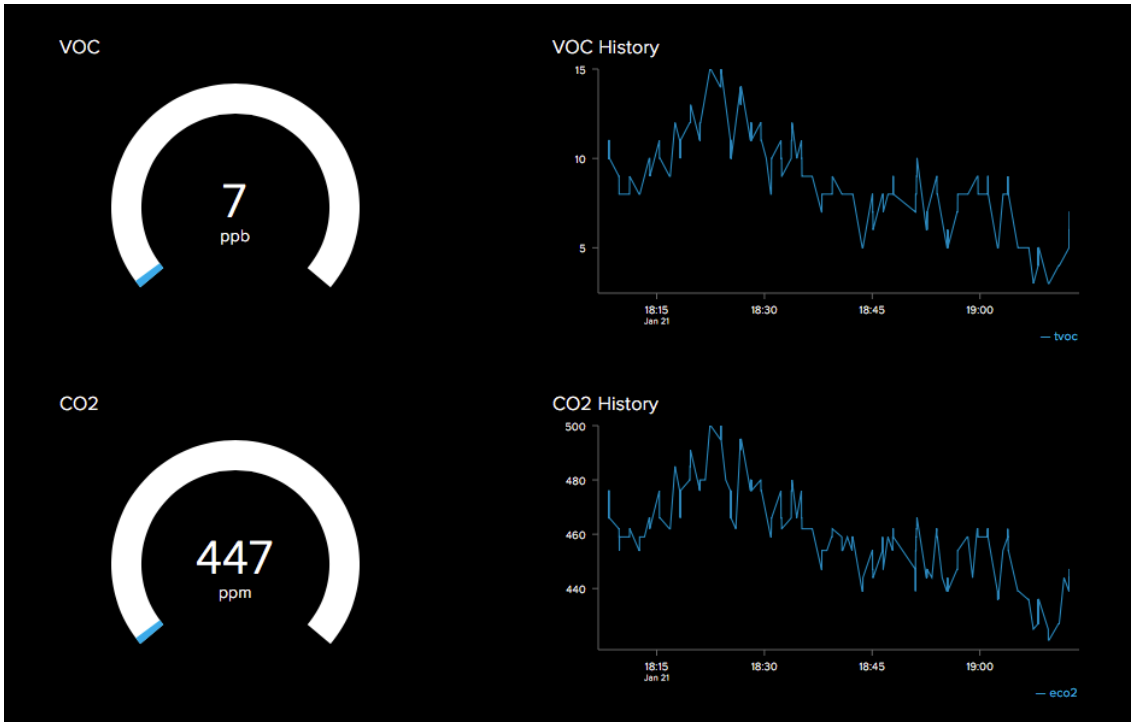



*Abbildung 156: Adafruit IO Dashboard Visualisierung der Daten (VOC und CO2)*

# 5. Erkenntnisse

Die Implementierung des MQTT-Testbettes stand im Vordergrund dieser Arbeit. Aus diesem Grund werden in diesem Kapitel Erkenntnisse, welche aus der Realisierung der Anforderungen entstanden, dargelegt. Dabei wird vorwiegend auf Besonderheiten und Schwierigkeiten eingegangen, die auftraten.

Anfangs galt es, die zentrale Stelle eines MQTT-Kommunikationsnetzes, den Broker auf dem Raspberry Pi einzurichten. Als MQTT-Broker-Implementierung wurde Mosquitto gewählt, da dies eine sehr weit verbreitete und beliebte Version einer Open Source Broker-Implementierung ist. Die Installation stellte sich für die aktuellste Version *Stretch* der Raspberry Distribution Raspbian aufgrund einer ungenügenden Dokumentation auf der Homepage der Organisation als schwierig heraus. Nach einiger Recherche in diversen Foren konnte das richtige Repository gefunden werden. Die erfolgreiche Installation wurde mithilfe einer Verbindung zu einem Client über das hilfreiche Debug-Tool MQTT.fx getestet.

Als nächstes stand die erstmalige Benutzung des Mikrocontrollers der aktuellsten ESP-Reihe, des ESP32 Development Boards an. Dafür musste die ESP32 Library für dieses Board in der Arduino IDE integriert werden. Dies erwies sich auf dem Mac als schwierig, da diese Bibliothek derzeit nur durch einen Workaround bezogen werden kann. Auch für die Installation der Treiber für den virtuellen Port bedurfte es einiger aufwändiger Recherche im Internet. Für das Einrichten des MQTT-Clients auf dem ESP32 waren jedoch lediglich ein paar Klicks nötig, um die Library in der Bibliotheksverwaltung der IDE zu beziehen. Hier waren direkt ein Verbindungsaufbau und der Nachrichtenaustausch mit dem Broker möglich, was mit MQTT.fx erfolgreich getestet werden konnte. Die Programmierung des Mikrocontrollers geschieht durch eine C-nahe Programmiersprache. Die Handhabung mit selbiger wird vorausgesetzt.

Schwierig gestaltete sich der erste Umgang mit dem UV-Sensor der Firma Adafruit. Dieser benutzt als Schnittstelle die I2C-Ports des ESP32 Development Boards. Es galt herauszufinden, welche Pins dort standardmäßig für die I2C-Kommunikation verwendet werden. Nachdem der Sensor über den durch Recherchen gefundenen Sketch zum Testen der I2C-Schnittstelle erkannt wurde, waren der Code für das Auslesen des Sensors und die Kommunikation mit dem Broker schnell geschrieben. Das war dank guter Code-Beispiele in der Bibliothek für den Sensor und den Client möglich. Einen beträchtlichen Aufwand benötigte die Suche nach einer Funktion, die die gemessenen Sensor-Werte in den offiziellen UV-Index umrechnet. Die Umrechnung erfolgt über eine komplizierte Formel. Diese fand sich schließlich in einem Forum. Danach wurde nach einer Subskription durch den MQTT.fx Client der aktuelle UV-Index angezeigt. Auffallend war, dass der Code gelegentlich nach dem Kompilieren aufgrund eines nicht ermittelbaren Fehlers nicht auf den ESP32 hochgeladen werden konnte. Der Fehler ließ sich durch mehrfaches Wiederholen des Upload-Prozesses beheben. Ebenfalls auffällig war, dass der Sketch auf dem Mikrocontroller gelegentlich nicht funktionierte, sondern erst nach mehrmaligem Drücken der Reset-Taste das Programm richtig startete.

Diese kleinen Probleme tauchten auch bei der Arbeit mit dem publizierenden ESP32 Client auf, der für die Messung der Luftqualität verantwortlich ist. Keine Probleme hingegen machte der ebenfalls über die I2C-Schnittstelle angeschlossene Luftqualität-Sensor.

Das Einrichten des empfangenden Clients mit dem NeoPixel Ring stellte sich durch die zuvor gewonnene Erfahrung mit dem ESP32 als problemlos heraus. Die mitgelieferte Bibliothek des LED-Rings ermöglichte ein schnelles Implementieren der aufleuchtenden Farb-Kombination bei einem bestimmten UV-Wert. Ebenfalls unproblematisch war die Implementierung des ESP32-Clients Fenster-Servomotors. Hier wurde beschlossen, Durchschnittswerte der CO2- und VOC-Messung über einen definierten Zeitraum zu überwachen, um ein ständiges Öffnen und Schließen des Fensters zu vermeiden. So reagiert der Aktor nur, wenn der Durchschnittswert den festgelegten Grenzwert über- und wieder unterschreitet. Diese Logik umzusetzen, war der schwierigste Teil dieser Aufgabe.

Die mobile App, die mit der visuellen Programmierumgebung Node-RED realisiert wurde und sämtliche Messdaten visualisiert, war innerhalb kurzer Zeit umgesetzt. Zwar handelt es sich hierbei um eine Web-App, aber es besteht die Möglichkeit, in kürzester Zeit flexibel Daten darstellen zu können. Die App soll exemplarisch zeigen, dass eine Integration von MQTT-Clients in Frameworks für App-Entwicklungen bereits stattgefunden hat.

Die Integration eines PaaS-Dienstes in dieses IoT-Testbett war eine obligatorische Anforderung. Hierbei ließ sich bei der Recherche nach verfügbaren Plattformen, die MQTT unterstützen, erkennen, dass viele Anbieter von plattformbetriebenen Diensten MQTT eine große Bedeutung zumessen. Hierbei gestaltete sich die Entscheidung zwischen der Auswahl der verschiedenen Anbieter schwierig. Ausgewählt wurde der weniger bekannte Dienst Adafruit IO, da dieser ein vielversprechendes Dashboard und vielfältige Features im kostenlosen Plan in Aussicht stellte. Im Nachhinein wurde festgestellt, dass dieser Dienst zwar leicht einzurichten, aber in der kostenlosen Version auf eine Rate ankommender Messages von sechzig pro Minute begrenzt war. Diese Rate wurde von den drei Clients im Testbett mit den insgesamt vier Topics beim Publizieren der Daten überschritten und führte zu einem teilweisen Blockieren der angezeigten Feeds.

Nachdem alles eingerichtet war, funktionierten die einzelnen Komponenten problemlos zusammen. Auch die anfänglichen Schwierigkeiten mit der I2C-Schnittstelle zu den Sensoren tauchten nicht wieder auf. Bei der Umsetzung der Implementierung konnten sämtliche Anforderungen wie geplant realisiert werden. Eine Vielzahl verfügbarer Literatur, Internet-Quellen, Dokumentationen sowie nützlicher Tools zum Thema MQTT ermöglichten es rückblickend sehr gut, ein derartiges Projekt umzusetzen. Durch die Verwendung aktuellster Hardware wie dem ESP32 Mikrocontroller und dem Raspberry Pi 3 Model B wurde zusätzlich geprüft, wie der Support und die Weiterentwicklung von Libraries und Software für die betreffenden neuen Geräte sind.

Auf diesem Bild wird der fertige Aufbau des MQTT-Testbettes ersichtlich. Auf der linken Seite befinden sich die zwei publizierenden EPS32-Clients mit UV- und Luftqualität-Sensor. In der Mitte läuft der auf dem Raspberry Pi installierte Mosquitto Broker. Rechts unten sind die subskribierenden ESP32-Clients mit LED-Farbring und Fenster-Servomotor angeordnet. Der mobile Web-App Client läuft auf einem iPad und visualisiert die Messwerte.

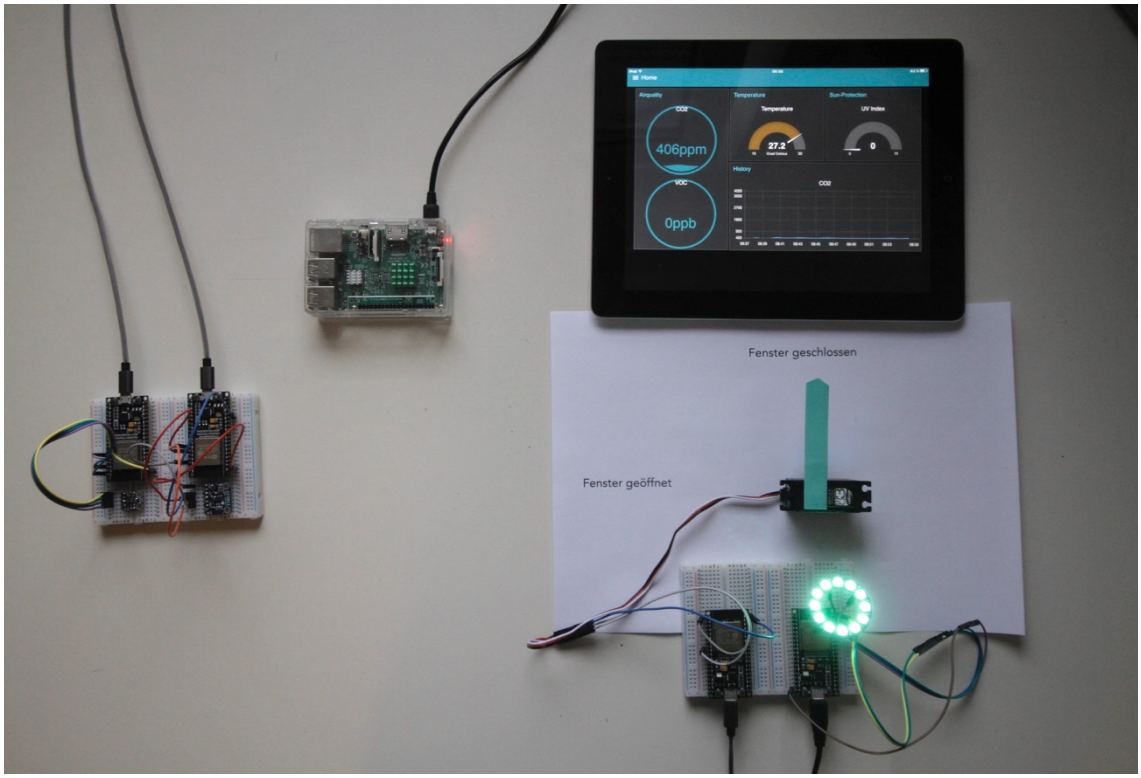



*Abbildung 157: Hardware der Anwendung im Betrieb*

# 6. Zusammenfassung und Fazit

Der Rahmen dieser Thesis umfasst die Untersuchung des Anwendungsprotokolls MQTT im Internet of Things. Die Untersuchung erfolgt in vier Abschnitten:

Zu Beginn leiten eine Definition und die Geschichte des Internet of Things in das Thema ein. Dabei zeigt sich, dass Forscher schon seit der Jahrtausendwende das Internet der Dinge untersuchen. Der Begriff des Ubiquitous Computing, also die Allgegenwärtigkeit von Computern, tauchte sogar schon zehn Jahre früher auf. Heutzutage zeigen sich viele Vorteile durch die Vernetzung von Computern mit physikalischen intelligenten Objekten. Prozessabläufe können in industriellen und urbanen Bereichen durch das Sammeln und Auswerten von Daten optimiert werden, intelligente Heim-Systeme erleichtern den Menschen das alltägliche Leben. Allerdings darf man einen wesentlichen Nachteil des IoT nicht vernachlässigen: Mit dem Internet vernetzte Geräte können zum Beispiel kompromittiert und die Daten missbraucht werden.

Für das Internet of Things bedarf es als Architektur grundlegend den Aufbau der Internetkommunikation mit den unterschiedlichen Protokollschichten. Auf der unteren Netzwerkschicht liegen Übertragungsprotokolle der verschiedenen Kommunikationskanäle. Diese Protokolle werden näher beschrieben, weil sie Grundlage für die Übertragung auf der Anwendungsschicht sind. Auf dieser obersten Schicht angeordnet befindet sich das Message Queue Telemetry Transfer, kurz MQTT. Die Aufgabe dieses Protokolls ist es, Nachrichten zwischen Schnittstellen von Anwendungen zu übermitteln.

Anschließend wird eine Recherche zum MQTT Protokoll durchgeführt. MQTT, vergleichbar mit HTTP, CoAP und XMPP, ist ein ursprünglich von IBM entwickeltes leichtgewichtiges Kommunikationsprotokoll für die M2M-Kommunikation im Internet of Things. Es wurde dafür entwickelt, um in limitierten Netzwerken eine zuverlässige Nachrichtenübertragung von Telemetrie-Daten zu gewährleisten. MQTT ist ein frei verfügbares Open-Source-Protokoll, das es seit 2014 als OASIS- und seit 2016 als ISO-Standard gibt. In der Funktionsweise unterscheidet sich das Protokoll von herkömmlichen Protokollen in der Architektur dahingehend, dass die Kommunikation anstatt mit dem klassischen Request-Response-Verfahren über die Publish/Subscribe-Architektur abgewickelt wird. Ein publizierender Client schickt nach dem Aufbau der Verbindung zum zentralen Server (Broker) diesem ereignisgesteuert Messdaten unter einem sogenannten Topic. Ein empfangender (subskribierender) Client, der an diesen Daten interessiert ist, abonniert nach dem Aufbau zum Broker das Topic, unter dem die relevanten Informationen an den Broker gesendet wurden. Daraufhin bekommt er vom Broker die Daten zugeschickt. MQTT wartet mit zusätzlichen Funktionen auf, die für den Bereich IoT von Bedeutung sind. So wird beispielsweise mit dem Feature Quality of Service dafür garantiert, dass Nachrichten beim Empfänger präzise ankommen, was gerade bei unzuverlässigen Netzen von großem Vorteil ist. Darüber hinaus sollten Daten in heutigen Anwendungen standardmäßig verschlüsselt übertragen werden: SSL-Verschlüsselung, Authentifizierung und Autorisierung sorgen bei Übertragungen mit sensiblen Daten bei MQTT für Sicherheit.

Gegenstand des dritten Teils dieser Arbeit ist die Konzipierung und Implementierung eines MQTT-Testbettes. Dazu gehört im Anschluss auch eine ausführliche Dokumentation, welche als Grundlage für eine Anleitung für ein Labor der Hochschule Offenburg genutzt werden kann. Zunächst werden die Anforderungen an das Testbett definiert: Ein MQTT-Broker auf einem Raspberry Pi ist für das Verwalten der Verbindungen der Clients und das Weiterleiten der Nachrichten verantwortlich. Implementierungen von MQTT-Clients, die auf einem ESP32 Mikrocontroller realisiert werden, publizieren ausgelesene Sensor-Daten an den Broker. Subskribierende Clients empfangen die konsolidierten Messwerte vom Broker. Auf einer als Client fungierenden mobilen App werden die Informationen visualisiert. Aktoren, mithilfe von empfangenden MQTT-Clients ebenfalls auf ESP32s realisiert, sollen auf bestimmte Werte Reaktionen ausführen. Ein PaaS-Dienst visualisiert die Messdaten zusätzlich. Die erste implementierte Anwendung befasst sich mit dem Erkennen von schädlichen UV-Strahlen. Ausgehend von einem UV-Sensor wird mithilfe einer App und einer LED-Lampe visuell farblich dargestellt, wie schädlich der aktuelle UV-Wert der Sonneneinstrahlung ist. So soll beispielsweise mithilfe der LED-Ampel daran erinnert werden, Sonnencreme mit einem entsprechenden Lichtschutzfaktor zu benutzen. Die Implementierung der zweiten Anwendung ist später Beispiel einer Labor-Aufgabe und wird hier exemplarisch durchgeführt. Hierbei handelt es sich um die Überwachung der Luftqualität in Innenräumen. Mithilfe eines Sensors für Luftqualität wird ebenfalls der gemessene Wert auf der App visualisiert. Beim Überschreiten eines Grenzwertes wird ein Fenstermotor in Gang gesetzt, um die Belüftung des Raumes zu ermöglichen.

Im letzten Abschnitt werden die Erkenntnisse aus dem realisierten Projekt dargelegt. Es lässt sich darauf bezugnehmend sagen, dass alle Anforderungen erfolgreich umgesetzt und auftretende Probleme behoben werden konnten.

Eingangs wurde MQTT als Protokoll vorgestellt, das die Eigenschaften eines sicheren, leichtgewichtigen, effizienten ereignisgesteuerten Übermittlungsprotokolls haben soll. Stimmen diese Eigenschaften mit den Anforderungen an ein IoT-Anwendungsprotokoll überein? Fakt ist, dass immer mehr Geräte mit dem Internet verbunden sein werden. Die begrenzte Bandbreite im Internet führt dazu, dass nach alternativen Protokollen gesucht werden muss, um das Netz nicht mit Paketen von Protokollen zu überlasten, dessen Overhead für den Anwendungsfall IoT zu hoch ist. MQTT arbeitet mit kleinen Datenpaketen, die sich sehr gut dafür eignen, Telemetrie-Daten, wie sie im IoT hauptsächlich vorkommen, zu verschicken. Damit wird eine effiziente Nutzung von Netzwerkressourcen erzielt.

Das MQTT zugrundeliegende Pub/Sub-Pattern sorgt durch dessen Architektur dafür, dass 1:n-Beziehungen bezüglich des Brokers möglich sind. Das hat eine sehr hohe Skalierbarkeit von Devices zur Folge, die miteinander kommunizieren können. Die Zuverlässigkeit und Flexibilität dieses IoT-Protokolls, so hat es sich bei der Implementierung dieses Projektes gezeigt, war jederzeit gewährleistet. Knotenpunkte, die spontan dem Netz entnommen wurden, konnten immer wieder problemlos hinzugefügt werden.

Aktuell werden, wie aufgezeigt, eine Vielzahl an Plattformen und Geräten, sowie unterschiedliche Datentypen, von textbasierten Daten bis zu Binärdateien, unterstützt. Es hat

sich bei den Recherchen herausgestellt, dass MQTT mittlerweile stark verbreitet ist. Global Player wie IBM und Amazon setzen das Protokoll bereits in ihren IoT-Services ein. Das alles, und der reibungslose Ablauf des Projekts, sprechen für eine Verwendung des Protokolls im Internet of Things. In Zukunft wird eine Auseinandersetzung mit MQTT in vielen Bereichen stattfinden. Wie eingangs erwähnt, ist IoT nicht auf Smart Home begrenzt. Ganze Städte werden sich zu Smart Cities wandeln. Die sich darin flexibel bewegenden Objekte werden alle miteinander vernetzt agieren. Auf die Herausforderung, dass weitere Milliarden von Devices über das Internet miteinander kommunizieren werden, muss reagiert werden. MQTT bietet ein Protokoll, das genau auf diesen Fall ausgelegt ist. Es ist damit zu rechnen, dass aus diesem Grund der Begriff MQTT im Bereich Internet of Things eine tragende Rolle einnehmen wird.

# 7. Literaturverzeichnis

[1] V. P. Andelfinger, Internet der Dinge - Technik, Trends und Geschäftsmodelle, T. Hänisch, Hrsg., Springer, 2015.

[2] M. Weiser, „The Computer for the 21st Century," *Scientific American*, pp. 94-104, 09 1991.

[3] IHS, „Statista - Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025," 2015/16. [Online]. Available: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/. [Zugriff am 09 11 2017].

[4] Cisco, „Cisco - The Internet of Things Reference Model," 2014. [Online]. Available: http://cdn.iotwf.com/resources/71/IoT_Reference_Model_White_Paper_June_4_2014.pdf. [Zugriff am 09 10 2017].

[5] M. G. M. M. M. A. M. A. A. Al-Fuqaha, „Internet of things: A survey on enabling technologies, protocols, and applications," in *IEEE COMMUNICATION SURVEYS & TUTORIALS, VOL. 17, NO. 4*, 2015, pp. 2347-2376.

[6] P. R. S. A. A. M. J. J. F. M. D. Miguel A. Prada, „Communication with resource-constrained devices through MQTT for control education," in *IFAC-PapersOnLine Volume 49, Issue 6, Pages 1-378 (2016)*, 1. I. S. o. A. i. C. E. A. 2016, Hrsg., Bratislava, 2016, pp. 150-155.

[7] S. Horvath, „Bundestag," 17 07 2012. [Online]. Available: https://www.bundestag.de/blob/192512/cfa9e76cdcf46f34a941298efa7e85c9/internet_der_dinge-data.pdf. [Zugriff am 07 10 2017].

[8] 6 10 2017. [Online]. Available: mqtt.org. [Zugriff am 6 10 2017].

[9] J. P. J. G. S. Hyun Cheon Hwang, „Design and Implementation of a Reliable Message Transmission System Based on MQTT Protocol in IoT," in *Wireless Personal Communications - An International Journal*, Volume 91 Hrsg., Bd. Issue 4, Springer, 2016, p. 1765–1777.

[10] P. Samulat, Die Digitalisierung der Welt Wie das Industrielle Internet der Dinge aus Produkten Services macht, Wiesbaden: Springer Gabler, 2016.

[11] D. Evans, „The Internet of Things - How the Next Evolution of the Internet Is Changing Everything," 2011.

[12] P. G. P. F. S. W. Harald Sundmaeker, Vision and Challenges for Realising the Internet of Things, C. o. E. R. P. o. t. I. o. Things, Hrsg., Publications Office of the European Union, 2010.

[13] S. Dodson, „The Guardian - "The Internet of Things"," 9 10 2003. [Online]. Available: https://www.theguardian.com/technology/2003/oct/09/shopping.newmedia. [Zugriff am 8 10 2017].

[14] S. Meloan, „Baidu - "Toward a Global Internet of Things"," 11 11 2003. [Online]. Available: https://wenku.baidu.com/view/16859b26ccbff121dd368320.html. [Zugriff am 08 10 2017].

[15] A. Reinhardt, „Bloomberg - "A Machine-to-Machine Internet of Things"," 26 04 2004. [Online]. Available: https://www.bloomberg.com/news/articles/2004-04-25/a-machine-to-machine-internet-of-things. [Zugriff am 8 10 2017].

[16] R. K. D. C. Neil Gershenfeld, „MIT - "The Internet of Things - The principles that gave rise to the Internet are now leading to a new kind of network of everyday devices."," 10 2004. [Online]. Available: http://cba.mit.edu/docs/papers/04.10.i0.pdf. [Zugriff am 08 10 2017].

[17] E. Union, „Europäische Union - IoT Aktionsplan KOM(2009) 278," 2009. [Online]. Available: http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=COM:2009:0278:FIN. [Zugriff am 16 10 2017].

[18] S. COBB, „welivesecurity - 10 things to know about the October 21 IoT DDoS attacks," 24 10 2016. [Online]. Available: https://www.welivesecurity.com/2016/10/24/10-things-know-october-21-iot-ddos-attacks/. [Zugriff am 16 10 2017].

[19] A. Kumar, „SearchSecurity - Internet der Dinge (IoT): Sieben wichtige Risiken für Unternehmen," 06 2014. [Online]. Available: http://www.searchsecurity.de/lernprogramm/Internet-der-Dinge-IoT-Sieben-Risiken-fuer-Firmen. [Zugriff am 16 10 2017].

[20] K. W. R. James F. Kurose, Computernetzwerke - Der Top-Down-Ansatz, 6. Auflage Hrsg., Pearson Studium, 2014.

[21] H. S. Christoph Meinel, Internetworking - Technische Grundlagen und Anwendungen, Berlin Heidelberg: Springer, 2012.

[22] C. Braun, Computernetze kompakt, 2. Auflage Hrsg., Berlin Heidelberg: Springer, 2012/13.

[23] R. Khan, S. U. Khan, R. Zaheer und S. Khan, „Future Internet: The Internet of Things Architecture, Possible Applications and Key Challenges," in *Frontiers of Information Technology (FIT)*, I. Conference, Hrsg., 2012.

[24] IoT-A, „The Internet-of-Things Architecture," [Online]. Available: http://www.meet-iot.eu/deliverables-IOTA/D1_3.pdf. [Zugriff am 09 10 2017].

[25] IoT-A, „IoTForum - Introduction to the Architectural Reference Model for the Internet of Things," [Online]. Available: http://iotforum.org/wp-content/uploads/2014/09/120613-IoT-A-ARM-Book-Introduction-v7.pdf. [Zugriff am 09 10 2017].

[26] T.-l. L. F.-Y. L. l. S. H.-Y. D. Miao Wu, „Research on the architecture of Internet of things," in *Proc. 3rd ICACTE*, 2010, p. 484–487.

[27] M. S. Gast, „802.11ac: A Survival Guide – Chapter 5. 802.11ac Planning," *Beamforming in 802.11ac*, 27 08 2013.

[28] N. Chen, Bluetooth Low Energy Based CoAP Communication in IoT, Saskatoon, 2016.

[29] B. S. Proprietary, Bluetooth Core Speci cation 5.0 FAQ, 2016.

[30] S. Hegenderfer, „Bluetooth Blog - Get ready for Bluetooth mesh!," 22 11 2016. [Online]. Available: https://blog.bluetooth.com/get-ready-for-mesh. [Zugriff am 10 11 2017].

[31] B. SIG, „Bluetooth - Mesh Model Specification," 13 07 2017. [Online]. Available: https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=429634. [Zugriff am 10 11 2017].

[32] R. K. Markus Kraußе, Drahtlose ZigBee-Netzwerke - Ein Kompendium, Wiesbaden: Springer Vieweg, 2014.

[33] T. Group, „Thread Group - Our Technology," [Online]. Available: https://threadgroup.org/technology/ourtechnology. [Zugriff am 26 10 2017].

[34] z-wavealliance, „z-wavealliance.org - About Z-Wave Technology," [Online]. Available: https://z-wavealliance.org/about_z-wave_technology/. [Zugriff am 26 10 2017].

[35] S. G. Behrang Fouladi, „Security Evaluation of the Z-Wave Wireless Protocol," in *Black hat USA*, 2013.

[36] E. Kompendium, „Elektronik Kompendium - EDGE - Enhanced Data Rates for GSM Evolution," [Online]. Available: http://www.elektronik-kompendium.de/sites/kom/0910171.htm. [Zugriff am 27 10 2017].

[37] LTE-Anbieter, „LTE-Anbieter - 5G: Alles zum LTE-Nachfolger der Zukunft," [Online]. Available: http://www.lte-anbieter.info/5g/. [Zugriff am 26 10 2017].

[38] E. Kompendium, „Elektronik Kompendium - NarrowBand-IoT (LTE-Cat-NB1)," [Online]. Available: https://www.elektronik-kompendium.de/sites/kom/2203161.htm. [Zugriff am 26 10 2017].

[39] L. Alliance, „LoRa Alliance," 2015. [Online]. Available: https://docs.wixstatic.com/ugd/eccc1a_ed71ea1cd969417493c74e4a13c55685.pdf. [Zugriff am 27 10 2017].

[40] W. Trojan, Das MQTT-Praxisbuch, Aachen: Elektor-Verlag GmbH, 2017.

[41] XMPP, „XMPP - An Overview of XMPP," [Online]. Available: https://xmpp.org/about/technology-overview.html. [Zugriff am 13 11 2017].

[42] XMPP, „XMPP - XEP-0060: Publish-Subscribe," 10 10 2017. [Online]. Available: https://xmpp.org/extensions/xep-0060.html. [Zugriff am 13 11 2017].

[43] W. C. K. S. G. M. G. R. Niccolò De Caro, „Comparison of two lightweight protocols for smartphone-based sensing," in *2013 IEEE 20th Symposium on Communications and Vehicular Technology in the Benelux (SCVT)*, 2013.

[44] O. S. Group, „Online Solutions Group - HTTP/2," [Online]. Available: https://www.online-solutions-group.de/online-marketing/seo-suchmaschinenoptimierung/seo-whitepaper/http2.html. [Zugriff am 10 11 2017].

[45] O. Open, „Oasis Open - Overview," [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=mqtt. [Zugriff am 29 10 2017].

[46] J. Dann, „Facebook - Under the hood: Rebuilding Facebook for iOS," 23 08 2012. [Online]. Available: https://www.facebook.com/notes/facebook-engineering/under-the-hood-rebuilding-facebook-for-ios/10151036091753920. [Zugriff am 06 11 2017].

[47] H. L. T. Andy Stanford-Clark, MQTT For Sensor Networks (MQTT-SN) - Protocol Specification, Bd. Version 1.2, IBM, Hrsg., 2013.

[48] A. A. A.-R. A. S. H. M. Tarek R. Sheltami, „A survey on developing publish/subscribe middleware over wireless sensor/actuator networks," in *Wireless Networks*, Issue 6 Hrsg., Bd. Volume 22, 2016, p. 2049–2070.

[49] J. S. Richard Okuniewicz, „MQTT - Message Queue Telemetry Transport Protocol," 2015.

[50] O. E. O. H. David Boswarthick, M2M Communications: A Systems Approach, Wiley, 2012.

[51] HiveMQ, „HiveMQ - Introducing MQTT," [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt. [Zugriff am 02 11 2017].

[52] ISO, „ISO - ISO/IEC 20922:2016," [Online]. Available: https://www.iso.org/standard/69466.html. [Zugriff am 02 11 2017].

[53] O. Open, „Oasis Open - MQTT Version 5.0," 13 07 2017. [Online]. Available: http://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf. [Zugriff am 29 10 2017].

[54] OASIS, „MQTT Version 3.1.1 OASIS Standard". 29 10 2014.

[55] HiveMQ, „HiveMQ - MQTT Essentials," [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials/. [Zugriff am 20 11 2017].

[56] HiveMQ, „HiveMQ - MQTT Essentials Part 2: Publish & Subscribe," [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe. [Zugriff am 06 11 2017].

[57] Y. W. H. L. Y. S. X. W. Z. W. Konglong Tang, „Design and Implementation of Push Notification System Based on the MQTT Protocol," in *The authors*, 2013.

[58] G. C. Hillar, MQTT Essentials - A lightweight IoT Protocol, Packt, 2017.

[59] HiveMQ, „HiveMQ - MQTT Essentials Part 5: MQTT Topics & Best Practices," [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices. [Zugriff am 14 11 2017].

[60] HiveMQ, „HiveMQ - MQTT Essentials Part 3: Client, Broker and Connection Establishment," [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment. [Zugriff am 14 11 2017].

[61] HiveMQ, „HiveMQ - MQTT Essentials Part 6: Quality of Service 0, 1 & 2," [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels. [Zugriff am 20 11 2017].

[62] HiveMQ, „HiveMQ - MQTT Essentials Part 4: MQTT Publish, Subscribe & Unsubscribe," [Online]. [Zugriff am 23 11 2017].

[63] HiveMQ, „HiveMQ - MQTT Essentials Part 7: Persistent Session and Queuing Messages," [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-7-persistent-session-queuing-messages. [Zugriff am 14 01 2018].

[64] HiveMQ, „HiveMQ - MQTT Essentials Part 8: Retained Messages," [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-8-retained-messages. [Zugriff am 28 11 2017].

[65] HiveMQ, „HiveMQ - MQTT Essentials Part 9: Last Will and Testament," [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-9-last-will-and-testament. [Zugriff am 14 01 2018].

[66] IBM, „IBM - Explore MQTT and the Internet of Things service on IBM Bluemix," [Online]. Available: https://www.ibm.com/developerworks/cloud/library/cl-mqtt-bluemix-iot-node-red-app/index.html. [Zugriff am 06 11 2017].

[67] Google, „Google - Google Trends," [Online]. Available: https://trends.google.com/trends/explore?date=today%205-y&q=mqtt. [Zugriff am 06 11 2017].

[68] D. Wetterdienst, „DWD - UV-Gefahrenindex," [Online]. Available: https://www.dwd.de/DE/leistungen/gefahrenindizesuvi/gefahrenindexuvi.html. [Zugriff am 01 11 2017].

[69] W. U. WHO, „WHO - Global Solar UV Index," [Online]. Available: http://www.who.int/uv/publications/en/UVIGuide.pdf. [Zugriff am 04 12 2017].

[70] senseBox:edu, „Github Sensebox - Dokumentation Open Educational Resources," [Online]. Available: https://sensebox.github.io/books/senseBox:edu_en.pdf. [Zugriff am 04 12 2017].

[71] T. Dack, „GitHubGist," [Online]. Available: https://gist.github.com/tdack/45dd356d9271a87914ce. [Zugriff am 21 01 2018].

[72] K. H. Torvmark, „EDN Network - Three flavors of Bluetooth: Which one to choose?," 29 01 2013. [Online]. Available: https://www.edn.com/Home/PrintView?contentItemId=4405960. [Zugriff am 23 10 2017].

# 8. Abbildungsverzeichnis

# 9. Anhang

## 9.1. Eigenständigkeitserklärung

Hiermit bestätige ich, dass die vorliegende Arbeit von mir selbstständig verfasst wurde und keine anderen als die angegebenen Hilfsmittel benutzt wurden. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken (dazu zählen auch Internetquellen) entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht.

Mario Sallat

## 9.2. Termine

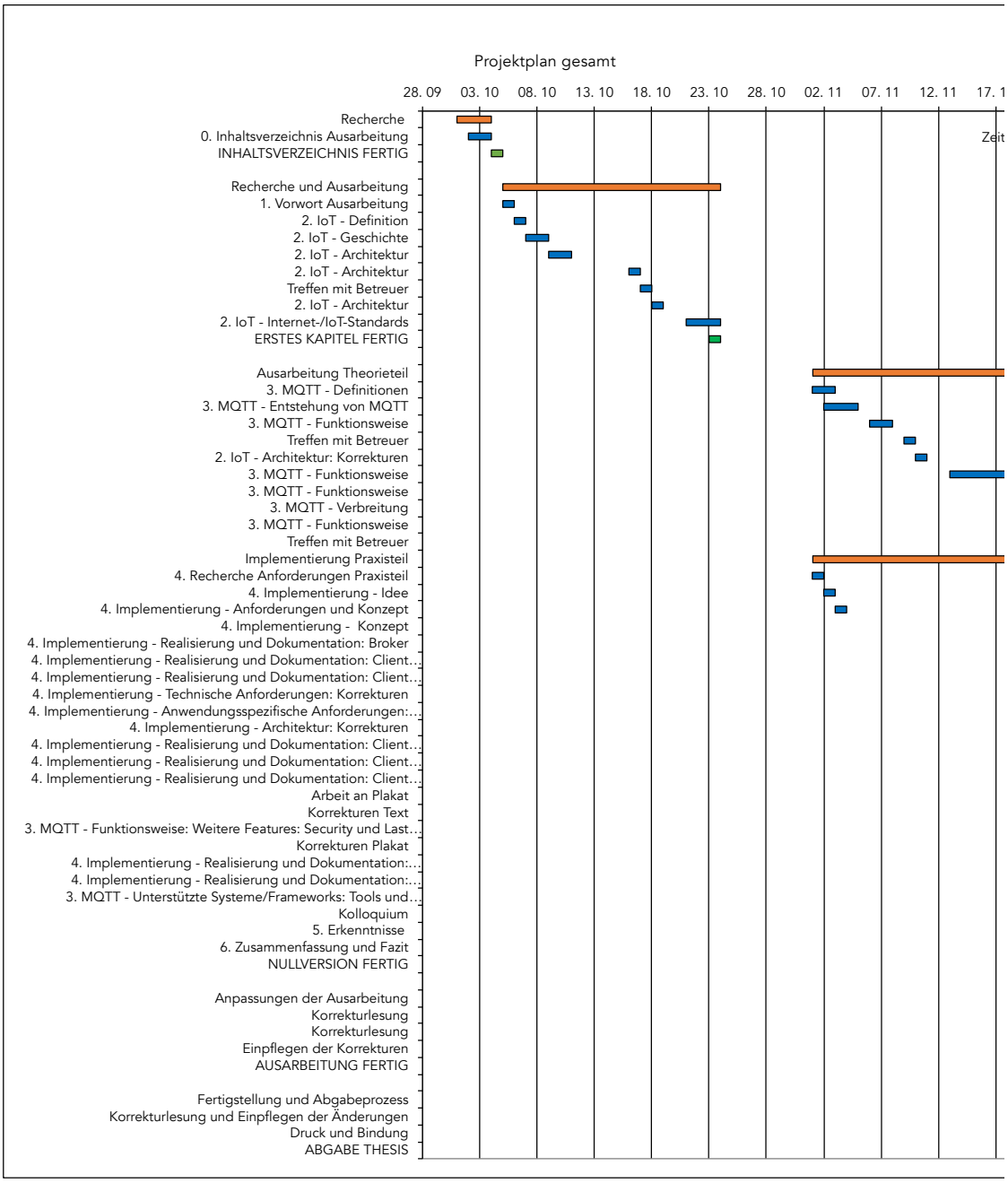
| | | | | |
|---|---|---|---|---|
| 1 | Recherche | 01.10.17 | 3 | 04.10.17 |
| 2 | 0. Inhaltsverzeichnis Ausarbeitung | 02.10.17 | 2 | 04.10.17 |
| 3 | INHALTSVERZEICHNIS FERTIG | 04.10.17 | 1 | 05.10.17 |
| 4 | Recherche und Ausarbeitung | 05.10.17 | 19 | 24.10.17 |
| 5 | 1. Vorwort Ausarbeitung | 05.10.17 | 1 | 06.10.17 |
| 6 | 2. IoT - Definition | 06.10.17 | 1 | 07.10.17 |
| 7 | 2. IoT - Geschichte | 07.10.17 | 2 | 09.10.17 |
| 8 | 2. IoT - Architektur | 09.10.17 | 2 | 11.10.17 |
| 9 | 2. IoT - Architektur | 16.10.17 | 1 | 17.10.17 |
| 10 | *Treffen mit Betreuer* | 17.10.17 | 1 | 18.10.17 |
| 11 | 2. IoT - Architektur | 18.10.17 | 1 | 19.10.17 |
| 12 | 2. IoT - Internet-/IoT-Standards | 21.10.17 | 3 | 24.10.17 |
| 14 | ERSTES KAPITEL FERTIG | 23.10.17 | 1 | 24.10.17 |
| 15 | Ausarbeitung Theorieteil | 01.11.17 | 83 | 23.01.18 |
| 16 | 3. MQTT - Definitionen | 01.11.17 | 2 | 03.11.17 |
| 17 | 3. MQTT - Entstehung von MQTT | 02.11.17 | 3 | 05.11.17 |
| 18 | 3. MQTT - Funktionsweise | 06.11.17 | 2 | 08.11.17 |
| 19 | *Treffen mit Betreuer* | 09.11.17 | 1 | 10.11.17 |
| 20 | 2. IoT - Architektur: *Korrekturen* | 10.11.17 | 1 | 11.11.17 |
| 21 | 3. MQTT - Funktionsweise | 13.11.17 | 6 | 19.11.17 |
| 22 | 3. MQTT - Funktionsweise | 21.11.17 | 2 | 23.11.17 |
| 23 | 3. MQTT - Verbreitung | 25.11.17 | 1 | 26.11.17 |
| 24 | 3. MQTT - Funktionsweise | 28.11.17 | 1 | 29.11.17 |
| 25 | *Treffen mit Betreuer* | 30.11.17 | 1 | 01.12.17 |
| 26 | Implementierung Praxisteil | 01.11.17 | 83 | 23.01.18 |
| 27 | 4. Recherche Anforderungen Praxisteil | 01.11.17 | 1 | 02.11.17 |
| 28 | 4. Implementierung - Idee | 02.11.17 | 1 | 03.11.17 |
| 29 | 4. Implementierung - Anforderungen und Konzept | 03.11.17 | 1 | 04.11.17 |
| 30 | 4. Implementierung - Konzept | 20.11.17 | 1 | 21.11.17 |
| 31 | 4. Implementierung - Realisierung und Dokumentation: Broker | 21.11.17 | 1 | 22.11.17 |
| 32 | 4. Implementierung - Realisierung und Dokumentation: Client Publisher 1 | 23.11.17 | 2 | 25.11.17 |
| 33 | 4. Implementierung - Realisierung und Dokumentation: Client Subscriber 1 | 26.11.17 | 2 | 28.11.17 |
| 34 | 4. Implementierung - Technische Anforderungen: *Korrekturen* | 01.12.17 | 1 | 02.12.17 |
| 35 | 4. Implementierung - Anwendungsspezifische Anforderungen: *Korrekturen* | 02.12.17 | 1 | 03.12.17 |
| 36 | 4. Implementierung - Architektur: *Korrekturen* | 02.12.17 | 1 | 03.12.17 |
| 37 | 4. Implementierung - Realisierung und Dokumentation: Client Subscriber 1 | 03.12.17 | 2 | 05.12.17 |
| 38 | 4. Implementierung - Realisierung und Dokumentation: Client Publisher 2 | 06.12.17 | 2 | 08.12.17 |
| 39 | 4. Implementierung - Realisierung und Dokumentation: Client Subscriber 2 | 09.12.17 | 3 | 12.12.17 |
| 40 | Arbeit an Plakat | 11.01.18 | 1 | 12.01.18 |
| 41 | Korrekturen Text | 13.01.18 | 2 | 15.01.18 |
| 42 | 3. MQTT - Funktionsweise: Weitere Features: Security und Last Will Testament | 14.01.18 | 1 | 15.01.18 |
| 43 | Korrekturen Plakat | 15.01.18 | 1 | 16.01.18 |
| 44 | 4. Implementierung - Realisierung und Dokumentation: Verwendete Hardware | 15.01.18 | 1 | 16.01.18 |
| 45 | 4. Implementierung - Realisierung und Dokumentation: Verwendete Software | 17.01.18 | 1 | 18.01.18 |
| 46 | 3. MQTT - Unterstützte Systeme/Frameworks: Tools und Programme - Apps | 18.01.18 | 3 | 21.01.18 |
| 47 | Kolloquium | 23.01.18 | 1 | 24.01.18 |
| 48 | 5. Erkenntnisse | 25.01.18 | 1 | 26.01.18 |
| 49 | 6. Zusammenfassung und Fazit | 26.01.18 | 1 | 27.01.18 |
| 50 | NULLVERSION FERTIG | 27.01.18 | 1 | 28.01.18 |

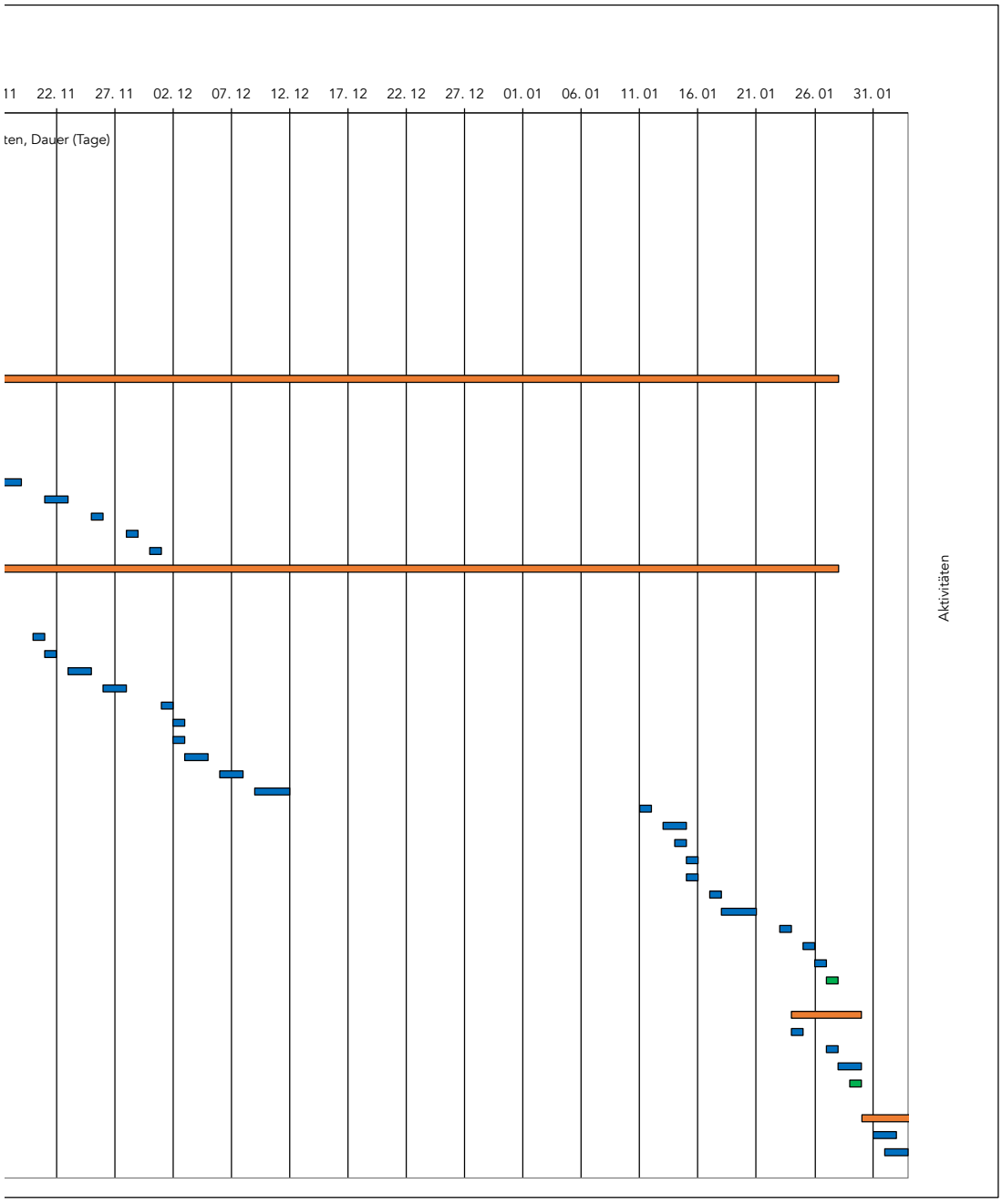
| | | | | |
|---|---|---|---|---|
| 51 | Anpassungen der Ausarbeitung | 24.01.18 | 6 | 30.01.18 |
| 52 | Korrekturlesung | 24.01.18 | 1 | 25.01.18 |
| 53 | Korrekturlesung | 27.01.18 | 1 | 28.01.18 |
| 54 | Einpflegen der Korrekturen | 28.01.18 | 2 | 30.01.18 |
| 55 | **AUSARBEITUNG FERTIG** | 29.01.18 | 1 | 30.01.18 |
| 56 | Fertigstellung und Abgabeprozess | 30.01.18 | 7 | 06.02.18 |
| 57 | Korrekturlesung und Einpflegen der Änderungen | 31.01.18 | 2 | 02.02.18 |
| 58 | Druck und Bindung | 01.02.18 | 2 | 03.02.18 |
| 62 | **ABGABE THESIS** | 05.02.18 | 1 | 06.02.18 |

Legende:

| |
|---|
| Meilensteine |
| Projektphase |
| Arbeitspakete |

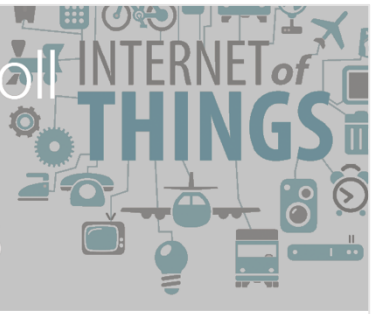
## 9.3. Projektplan



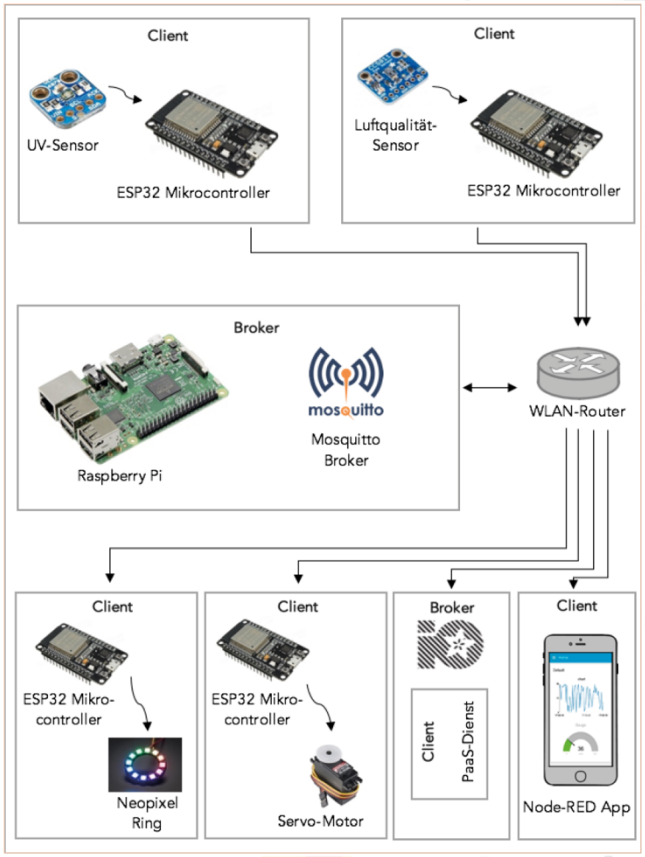

11
22. 11
27. 11
02. 12
07. 12
12. 12
17. 12
22. 12
27. 12
01. 01
06. 01
11. 01
16. 01
21. 01
26. 01
31. 01
ten, Dauer (Tage)
Aktivitäten

## 9.4. Plakat des Kolloqiums

# Das Anwendungsprotokoll **MQTT** im **INTERNET OF THINGS**




Gesamtarchitektur der Anwendung:




**Inhalt der Arbeit:**

- Recherche zu MQTT
- Anforderungen an das MQTT-Testbett:
  - Broker
  - publizierende Clients
  - subskribierende Clients
  - mobile App
  - PaaS-Dienst
  - Exempl. Anwendung
- Konzept
- Implementierung
- Dokumentation
- Bewertung von MQTT

Bachelor Thesis im WS 2017/2018
im Studiengang
Medien und Informationswesen

von Mario Sallat
betreut durch
Prof. Dr. Tom Rüdebusch

Hochschule Offenburg
offenburg.university

M+I Medien und Informationswesen

## 9.5. Code

client_test

```
/*
* client_test
*/
#include <WiFi.h>
#include <PubSubClient.h>

const char* ssid = "WIFI_SSID";
const char* password = "WIFI_PASSWORD";
const char* mqttServer = "10.0.1.10";
const int mqttPort = 1883;
const char* mqttUser = "user1";
const char* mqttPassword = "userpassword1";

WiFiClient espClient;
PubSubClient client(espClient);

void setup() {

  Serial.begin(115200);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.println("Connecting to WiFi..");
  }

  Serial.println("Connected to the WiFi network");

  client.setServer(mqttServer, mqttPort);

  while (!client.connected()) {
    Serial.println("Connecting to MQTT...");

    if (client.connect("ESP32Client", mqttUser, mqttPassword )) {

      Serial.println("connected");

    } else {

      Serial.print("failed with state ");
      Serial.print(client.state());
      delay(2000);

    }
  }

  client.publish("esp/test", "Hello from ESP32");

}

void loop() {
  client.loop();
}
```

i2c_scanner

```
/*
* i2c_scanner
* Scans ports and searches for I2C device. based on the original code
* available on Arduino.cc and later improved by user Krodal and Nick
Gammon (www.gammon.com.au/forum/?id=10896)
*
*/

#include <Wire.h>

void setup() {
  Serial.begin(115200);
  while (!Serial);             // Leonardo: wait for serial monitor
  Serial.println("\n\nI2C Scanner to scan for devices on each port pair
D0 to D7");
  scanPorts();
}

uint8_t portArray[] = {21, 22, 32, 33};

String portMap[] = {"GPIO21", "GPIO22", "GPI32", "GPIO33"};

void scanPorts() {
  for (uint8_t i = 0; i < sizeof(portArray); i++) {
    for (uint8_t j = 0; j < sizeof(portArray); j++) {
      if (i != j){
         Serial.print("Scanning (SDA : SCL) - " + portMap[i] + " : " +
portMap[j] + " - ");
        Wire.begin(portArray[i], portArray[j]);
        check_if_exist_I2C();
      }
    }
  }
}

void check_if_exist_I2C() {
  byte error, address;
  int nDevices;
  nDevices = 0;
  for (address = 1; address < 127; address++ )  {
    // The i2c_scanner uses the return value of
    // the Write.endTransmisstion to see if
    // a device did acknowledge to the address.
    Wire.beginTransmission(address);
    error = Wire.endTransmission();

    if (error == 0){
      Serial.print("I2C device found at address 0x");
      if (address < 16)
        Serial.print("0");
      Serial.print(address, HEX);
      Serial.println("  !");

      nDevices++;
    } else if (error == 4) {
      Serial.print("Unknow error at address 0x");
      if (address < 16)
        Serial.print("0");
      Serial.println(address, HEX);
    }
  } //for loop
  if (nDevices == 0)
```

```
    Serial.println("No I2C devices found");
  else
    Serial.println("**********************************\n");
  //delay(1000);          // wait 1 seconds for next scan, did not find
it necessary
}

void loop() {
}
```

client_uv_sensor

```
/*
* client_uv_sensor
*/
#include <WiFi.h>
#include <PubSubClient.h>

#include <Wire.h>
#include "Adafruit_VEML6070.h"

const char* ssid = "WIFI_SSID";
const char* password = "WIFI_PASSWORD";
const char* mqttServer = "10.0.1.10";
const int mqttPort = 1883;
const char* mqttUser = "user1";
const char* mqttPassword = "userpassword1";
char* topic = "garden/uvsensor";

// reference value: 0.01 W/m^2 corresponds to the UV-index 0.4
const float refVal = 0.4;

char payload[6];
String payload_str;

WiFiClient espClient;
PubSubClient client(espClient);

Adafruit_VEML6070 uv = Adafruit_VEML6070();

void setup() {
  Serial.begin(115200);
  setup_wifi();
  uv.begin(VEML6070_1_T);  // pass in the integration time constant
  client.setServer(mqttServer, mqttPort);
}

void loop() {

  // MQTT connection
  if (!client.connected()) {
    reconnectMQTT();
  }
  client.loop();

  //read sensor and get value as UV-Index
  uint32_t uvi = getUVI(uv.readUV());
  Serial.print("Sensor raw data: ");
  Serial.println(uvi);

  // preparing string
  payload_str = String(uvi);
  payload_str.toCharArray(payload, payload_str.length() + 1 );
```

```
  // publish
  publishToTopic(payload, topic);
}

/*
* getUVI()
* expects the measurement value from the UV-sensor as input * and returns
the corresponding value on the UV-index
*/
int getUVI(uint32_t uv) {
  float uvi = refVal * (uv * 5.625) / 1000;
  return (int)uvi;
}

/*
* setup_wifi()
* Establishment of the wifi connection
*/
void setup_wifi() {
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.println("Connecting to WiFi..");
  }

  Serial.println("Connected to the WiFi network");
}

/*
* reconnectMQTT()
* Connection establishment to the Broker
*/
void reconnectMQTT() {
  while (!client.connected()) {
    Serial.println("Connecting to MQTT...");

    // Create a random client ID
    String clientId = "ESP32Client-";
    clientId += String(random(0xffff), HEX);

    if (client.connect(clientId.c_str(), mqttUser, mqttPassword )) {

      Serial.println("connected");
    } else {

      Serial.print("failed with state ");
      Serial.print(client.state());
      delay(2000);
    }
  }
}

/*
* publishToTopic()
* publish to a specified topic
*/
void publishToTopic(char* payload, char* topic){
  if (client.connected()){
    Serial.print("Sending payload: ");
    Serial.print(payload);
    Serial.print(" to topic: ");
```

```
    Serial.println(topic);

    if (client.publish(topic, payload)) {
        Serial.println("Publish ok");
    }
    else {
      Serial.println("Publish failed");
    }
    Serial.println();
  }
}
```

client_neopixel

```
/*
* client_neopixel
*/
#include <WiFi.h>
#include <PubSubClient.h>
#include <Wire.h>
#include <Adafruit_NeoPixel.h>

#define PIN 22
#define NUM_LEDS 12
#define BRIGHTNESS 100

Adafruit_NeoPixel strip = Adafruit_NeoPixel(NUM_LEDS, PIN, NEO_GRB +
NEO_KHZ800);

const char* ssid = "WIFI_SSID";
const char* password = "WIFI_PASSWORD";
const char* mqttServer = "10.0.1.10";
const int mqttPort = 1883;
const char* mqttUser = "user1";
const char* mqttPassword = "userpassword1";
char* topic = "garden/uvsensor";

const uint32_t purple = strip.Color(134, 111, 255, 50);
const uint32_t red = strip.Color(248, 17, 22, 50);
const uint32_t orange = strip.Color(251, 116, 27, 50);
const uint32_t yellow = strip.Color(252, 199, 33, 50);
const uint32_t green = strip.Color(67, 185, 30, 50);
const int animationTime = 50;

uint32_t uvi = 1;
uint32_t uviTemp = 2;
uint32_t currentColor;

WiFiClient espClient;
PubSubClient client(espClient);

void setup() {
  Serial.begin(115200);
  setup_wifi();
  client.setServer(mqttServer, mqttPort);
  client.setCallback(callback);

  strip.setBrightness(BRIGHTNESS);
  strip.begin();
  // Initialize all pixels to 'off'
  strip.show();
  // set all pixels dark
  resetColor(0);
```

```
}

void loop() {
  // MQTT connection
  if (!client.connected()) {
    reconnectMQTT();
  }
  client.loop();

  // if the current uv-index changes, change color
  if (uvi != uviTemp){
    Serial.println("uv-index changed");
    uvi = uviTemp;

    // reset color
    resetColor(animationTime-20);

    if (uviTemp <= 2){
      currentColor = green;
    } else if (uviTemp <= 5){
      currentColor = yellow;
    } else if (uviTemp <= 7){
      currentColor = orange;
    } else if (uviTemp <= 10){
      currentColor = red;
    } else if (uviTemp >= 11){
      currentColor = purple;
    }
    // set current color
    colorWipe(currentColor, animationTime);
    Serial.println();
  }
}

void callback(char* topic, byte* payload, unsigned int length) {
    Serial.print("Message arrived [");
    Serial.print(topic);
    Serial.println("] ");

    String message;
    for (int i = 0; i < length; i++) {
      // prepare payload message
      message+=(char)payload[i];
    }

    // convert message to integer
    uviTemp = message.toInt();
    Serial.print("UV-Index: ");
    Serial.println(uviTemp);
}


/*
* setup_wifi()
* Establishment of the wifi connection
*/
void setup_wifi() {
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.println("Connecting to WiFi..");
  }
```

```
  Serial.println("Connected to the WiFi network");
}

/*
* reconnectMQTT()
* Connection establishment to the Broker
*/
void reconnectMQTT() {
  while (!client.connected()) {
    Serial.println("Connecting to MQTT...");

    // Create a random client ID
    String clientId = "ESP32Client-";
    clientId += String(random(0xffff), HEX);

    if (client.connect(clientId.c_str(), mqttUser, mqttPassword )) {

      Serial.println("connected");

      // subscribe to topic
      client.subscribe(topic);

    } else {

      Serial.print("failed with state ");
      Serial.print(client.state());
      delay(2000);
    }
  }
}

/*
* colorWipe()
* Fill the dots one after the other with a color
*/
void colorWipe(uint32_t c, uint8_t wait) {
  for(uint16_t i=0; i<strip.numPixels(); i++) {
    strip.setPixelColor(i, c);
    strip.show();
    delay(wait);
  }
}

/*
* resetColor()
* Reset all dots after the other with no color
*/
void resetColor(int animationTime){
  colorWipe(strip.Color(0, 0, 0), animationTime);
}
```

client_airquality_sensor

```
/*
* client_airquality_sensor
*/
#include <WiFi.h>
#include <PubSubClient.h>
#include "Adafruit_CCS811.h"

const char* ssid = "WIFI_SSID";
const char* password = "WIFI_PASSWORD";
const char* mqttServer = "10.0.1.10";
const int mqttPort = 1883;
const char* mqttUser = "user1";
const char* mqttPassword = "userpassword1";

WiFiClient espClient;
PubSubClient client(espClient);

Adafruit_CCS811 ccs;

void setup() {
  Serial.begin(115200);
  setup_wifi();
  client.setServer(mqttServer, mqttPort);

  if(!ccs.begin()){
    Serial.println("Failed to start sensor! check your wiring.");
    while(1);
  }

  //calibrate temperature sensor
  while(!ccs.available());
  float temp = ccs.calculateTemperature();
  ccs.setTempOffset(temp - 25.0);
}

void loop() {
  char payload[6];
  String payload_str;
  // MQTT connection
  if (!client.connected()) {
    reconnectMQTT();
  }
  client.loop();

  // read sensor data
  if(ccs.available()){
    if(!ccs.readData()){

      // read CO2
      int eCO2 = ccs.geteCO2();
      //prepare char
      payload_str = String(eCO2);
      payload_str.toCharArray(payload, payload_str.length() + 1 );
      // publish CO2
      publishToTopic(payload, "room/airquality/co2");

      // read TVOC
      int tvOC = ccs.getTVOC();
      // prepare char
      payload_str = String(tvOC);
      payload_str.toCharArray(payload, payload_str.length() + 1 );
      // publish TVOC
```

```
      publishToTopic(payload, "room/airquality/tvoc");

      // read temperature
      float temperature = ccs.calculateTemperature();
      // prepare char
      payload_str = String(temperature);
      payload_str.toCharArray(payload, payload_str.length() + 1 );
      // publish temperature
      publishToTopic(payload, "room/airquality/temperature");
    }
    else{
      Serial.println("ERROR!");
      while(1);
    }
  }

  delay(1000);
}

/*
* setup_wifi()
* Establishment of the wifi connection
*/
void setup_wifi() {
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.println("Connecting to WiFi..");
  }

  Serial.println("Connected to the WiFi network");
}

/*
* reconnectMQTT()
* Connection establishment to the Broker
*/
void reconnectMQTT() {
  while (!client.connected()) {
    Serial.println("Connecting to MQTT...");

    // Create a random client ID
    String clientId = "ESP32Client-";
    clientId += String(random(0xffff), HEX);

    if (client.connect(clientId.c_str(), mqttUser, mqttPassword )) {

      Serial.println("connected");
    } else {

      Serial.print("failed with state ");
      Serial.print(client.state());
      delay(2000);
    }
  }
}

/*
* publishToTopic()
* publish to a specified topic
*/
void publishToTopic(char* payload, char* topic){
```

```
  if (client.connected()){
    Serial.print("Sending payload: ");
    Serial.print(payload);
    Serial.print(" to topic: ");
    Serial.println(topic);

    if (client.publish(topic, payload)) {
        Serial.println("Publish ok");
    }
    else {
      Serial.println("Publish failed");
    }
    Serial.println();
  }
}
```

client_servo_motor

```
/*
* client_servo_motor
*/

#include <WiFi.h>
#include <PubSubClient.h>
#include <Wire.h>
#include <ESP32_Servo.h>

const char* ssid = "WIFI_SSID";
const char* password = "WIFI_PASSWORD";
const char* mqttServer = "10.0.1.10";
const int mqttPort = 1883;
const char* mqttUser = "user1";
const char* mqttPassword = "userpassword1";

const int measurementPeriod = 10;

char* tvOCtopic = "room/airquality/tvoc";
int eCO2Values[measurementPeriod+10];
int eC02PeriodIndex = 0;
const int eCO2Threshold = 600;

char* eCO2Topic = "room/airquality/eco2";
int tvOCValues[measurementPeriod+10];
int tvOCPeriodIndex = 0;
const int tvOCThreshold = 25;

int servoPin = 25;
int isWindowOpen = false;

Servo myservo;

WiFiClient espClient;
PubSubClient client(espClient);

void setup() {
  Serial.begin(115200);
  myservo.attach(servoPin);
  setup_wifi();
  client.setServer(mqttServer, mqttPort);
  client.set-
Callback(callback);
}
```

```
void loop() {
  // MQTT connection
  if (!client.connected()) {
    reconnectMQTT();
  }
  client.loop();

  if (eC02PeriodIndex == measurementPeriod){
    if (calculateAverage(eCO2Values) > eCO2Threshold &&
        calculateAverage(tvOCValues) > tvOCThreshold &&
        isWindowOpen == false){
            openWindow(true);
    } else if (calculateAverage(eCO2Values) < eCO2Threshold &&
               calculateAverage(tvOCValues) < tvOCThreshold &&
               isWindowOpen == true){
            openWindow(false);
    }
    eC02PeriodIndex = 0;
    tvOCPeriodIndex = 0;
    memset(eCO2Values, 0, sizeof eCO2Values);
    memset(tvOCValues, 0, sizeof tvOCValues);
  }
}

void callback(char* topic, byte* payload, unsigned int length) {
    Serial.print("Message arrived [");
    Serial.print(topic);
    Serial.println("] ");

    String message;
    for (int i = 0; i < length; i++) {
      // prepare payload message
      message+=(char)payload[i];
    }

    // tvOC handling
    if(strcmp(topic, tvOCtopic) == 0){
      tvOCValues[tvOCPeriodIndex] = message.toInt();
      Serial.print("Value: ");
      Serial.println(tvOCValues[tvOCPeriodIndex]);
      tvOCPeriodIndex++;
    }

    // eCO2 handling
    else if(strcmp(topic, eCO2Topic) == 0){
      eCO2Values[eC02PeriodIndex] = message.toInt();
      Serial.print("Value: ");
      Serial.println(eCO2Values[eC02PeriodIndex]);
      eC02PeriodIndex++;
    }
}


/*
* setup_wifi()
* Establishment of the wifi connection
*/
void setup_wifi() {
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.println("Connecting to WiFi..");
```

```
  }

  Serial.println("Connected to the WiFi network");
}

/*
* reconnectMQTT()
* Connection establishment to the Broker
*/
void reconnectMQTT() {
  while (!client.connected()) {
    Serial.println("Connecting to MQTT...");

    // Create a random client ID
    String clientId = "ESP32Client-";
    clientId += String(random(0xffff), HEX);

    if (client.connect(clientId.c_str(), mqttUser, mqttPassword )) {

      Serial.println("connected");

      // subscribe to topics
      client.subscribe(eCO2Topic);
      client.subscribe(tvOCtopic);
    } else {

      Serial.print("failed with state ");
      Serial.print(client.state());
      delay(2000);
    }
  }
}

/*
* calculateAverage()
* input parameter is the history of the last values
* returns average value
*/
float calculateAverage(int* arrayOfValues){
  int sum = 0;
  for (int i = 0; i < sizeof(arrayOfValues); i++) {
      sum += arrayOfValues[i];
  }
  Serial.print("average: ");
  Serial.println(sum/sizeof(arrayOfValues));
  return sum/sizeof(arrayOfValues);
}

/*
* openWindow()
* input parameter defines the status of the window
*/
void openWindow(boolean stat){
  if (stat == true){
    Serial.println("********* open window **********");
    isWindowOpen = true;
    for (int pos = 0; pos <= 180; pos += 1) {

      myservo.write(pos);
      delay(15);
    }
  } else {
    Serial.println("********** close window ***********");
```

```
    isWindowOpen = false;
    for (int pos = 180; pos >= 0; pos -= 1) {
      myservo.write(pos);
      delay(15);
    }
  }

}
```